Disk BASIC入門

大野・飯田・鈴木　共著

アスキー・ラボラトリーズ監修

Disk-BASIC入門

飯田　陽一　鈴木　義明
大野　俊治　共著
アスキー・ラボラトリーズ監修

アスキー出版

目次

0 ディスクはいったい何のため

「ディスクなんて，いったい何のためにあるんだろう？　カセットテープがあるからいいじゃないか？」と，お思いの方はいませんか．そうです．カセットテープと同じものだったら，あるいは，ちょっとしか違わないものだったら，何もディスクなんて買う必要はないのです．実際，PC-8001専用のディスク装置PC-8031は，値段が31万円（PC-8001本体の何と約2倍！）．ディスクがどうしてそんな価値があるのかと，不思議に思われるのはごく当然のことです．この本では，この問題「ディスクはどうして必要か？」を考えていこうと思います．

この本を一冊読み終えたとき，きっとあなたはディスクがほしくなってくるに違いありません．つまりディスクが単なる情報の保存装置ではなく，巨大なソフトウエアのかたまりであり，マイコンを一応ミニコン並みのレベルに引きあげるものであることが，きっとおわかりになるに違いないのです．

さて，まずこの章ではディスクのハードウエアのもつメリットとデメリットを考えていきましょう．

メリット

フロッピーディスクシステムのメリットは

- 動作が速い！
- 記憶容量が大きい！
- ランダムアクセスができる！

の3点であります．以下にそれぞれのメリットについて解説しますが，注意してほしいのは個々のメリットがあるためにフロッピーシステムが便利であるというよりも，3つのメリットがそろって全体として本当に便利といえるシステムになっているということです．

メリットその1　動作が速い！

単純にフロッピーディスクのディスケット面への読み書きの時間だけを考えてみましょう．公表されているデータによれば，4096バイトの情報の読み書きに必要な時間は最小で0.3秒，最大

で1.82秒だとされています．

この時間は，ディスケット面の読み書きをするヘッドが移動するための時間を含んでいます．つまり最小時間というのは，ヘッドが移動せずに読み書きを行った時の時間で，最大時間というのはヘッドがディスケットの端から端まで移動して読み書きする時間です．(実際の使用ではヘッドがディスケットの端から端まで移動するということはめったにありません．)Disk BASICではフロッピーディスクシステムを効果的に利用するためにディスケット上にいろいろな情報を載せていて，これを参照しながら読み書きをしますからもう少し時間がかかるでしょう．(この点については，第3章で詳しく説明します．)

メリットその2　記憶容量が大きい！

さて，ディスケット1枚には何バイトのデータを記録できるのでしょうか？

答え143.36Kバイトです，同時に2枚のディスケットを使用できますから，ユニット当りで286.72Kバイト記録できることになります．が，先にも書いたように，ディスケット上には，システムを効果的に使用するための情報も記録されていますから286.72Kバイトすべて使用者が使うことはできません．システム用の情報量は4,096バイトですが，PC-8031 (8032) に付いてるシステムディスクをコピーしたディスクはさらに12,288バイト必要なので合計で16,384バイトがシステム用の情報量となります．従って使用者が使える容量は

システムディスクで 126.976K バイト

その他のディスクで 139.264K バイト

になります．

この容量を大容量とするか，少容量とするかはあなたの考え方次第ですが，一本のプログラム，一組のデータの集合を記録するのに容量が足りなくなることはまずないでしょう．

メリットその3　ランダムアクセス方式ができる！

ランダムアクセス可能ということは先に書いた2つのメリットと密接に関係しているのですが，その説明をする前に「ランダムアクセス」とは何かを説明しましょう．ついでに「シーケンシャルアクセス」についても説明します．

まず「アクセス」とは，多量のデータの中から目的とするデータを選び出すことです．ディスクの場合にはディスケット上の目的の場所を読むとか，目的の場所に書きこむ動作が「アクセス」にあたるわけです．

さて，「アクセス」を知ったところでランダムアクセス方式とシーケンシャルアクセス方式の説明に入りましょう．まず身近な音楽の例から紹介します．いくつもの歌が録音されたレコードとカセットテープがあるとします．ランダムアクセスとはレコードの中から好きな曲を1つ選ぶこと，シーケンシャルアクセスとはテープの中から好きな曲を1つ選ぶことに相当します．こう書いただけでは「同じじゃないか」と思われるかも知れません．レコードとテープは，いったいどこが本質的に異なっているのでしょうか？

レコードの曲を選ぶにはレコード針を望みの曲の前に置けばいいのですが，ここで重要なことは針を置く場所は自由だということです．だからレコードに録音されている曲は最初の曲からでも，最後の曲からでも自由に聴くことができます．

一方，カセットテープは曲を選ぶ時にはその曲がテープの先頭にない時には，PLAYボタンの前にF.F.ボタンを押してテープを望みの曲まで進める手順が必要不可欠です．

なんとなくわかってきたのではないでしょうか？ レコードの曲はレコード上の内側にあろうと外側にあろうと直接アクセスできるのです．このようなアクセス方式をランダムアクセスと言います．カセットは直接アクセスできるのは

次の一曲だけで，その他の曲をアクセスするためには望みの曲までガマンして聞くか，途中の曲をとばさなければなりません．このようなアクセス方式をシーケンシャルアクセスと言います．

ところでランダムアクセスとシーケンシャルアクセスの説明のためにここまではレコードとカセットテープ中の音楽で話を進めてきましたが，ディスクのアクセス方法もレコードと殆ど同じなのです．ただレコードで針を動かす作業にあたるヘッドの移動は PC-8031 (8032) が自動的にやりますが…．さらにカセット中の音楽はカセット中のプログラムやデータと考えることができます．もう一度レコードをディスクに，プログラム・データ用カセットを音楽用カセットに，音楽をプログラムやデータに読みかえながらランダムアクセスとシーケンシャルアクセスの説明を読んでみてください．

ランダムアクセスは，多数のデータ中から任意のデータを選択できるアクセス方法で，シーケンシャルアクセスは，多数のデータはある特定の順序でしか選択できないアクセス方法です．

長々とランダムアクセスについて説明してきましたが，ランダムアクセス方式とシーケンシャルアクセス方式を理解すれば，何故にランダムアクセス方式がディスクのメリットなのかも理解しやすいはずです．

***ランダムアクセス方式のメリット**

ランダムアクセス方式のメリットはランダムアクセス方式とシーケンシャルアクセス方式の違いを比較すればわかってきます．目的のデータを得るまでの時間が短いということです．ランダムアクセス方式は目的とするデータを直接選択できますが，シーケンシャルアクセス方式は目的のデータを得るまで順々に読み出す動作が必要ですから，どちらが速いかは明らかでしょう．

先にディスクのメリットは速くて大容量なことだと書きましたが，よく考えてみてください．大容量というメリットと高速というメリットはある程度背反するとは思いませんか？たしかにカセットのようにシーケンシャルアクセス方式の記録媒体では，多数のデータを記録すれば最後に記録されたデータのアクセスには時間がかかりますが，ランダムアクセス方式の場合にはデータを多数記録しても，それぞれのデータのアクセスに要する時間はほとんど一定になります．すなわち，ランダムアクセス方式はそれ自体がメリットというよりも，高速で大容量という2つのメリットを実現する手段なのです．

ディスクのディメリット

ここまではディスクのメリットについて書いてきたのですが，公正を期すためにもディスクのディメリットについて書きます．下にディスクのディメリットを項目別に書いて，それについて説明します．心を落ち着けて読んでください．

- ・アクセスが遅い！
- ・記憶容量が小さい！
- ・複雑なソフトウエアが必要！
- ・弱い！

以上の4つがディスクのディメリットです．メリットの項に書いたことと全く逆の項目もありますが，別に嘘を書いているわけではありません．その理由を個々の項目について説明しましょう．

ディメリットその1　アクセスが遅い！

ディメリットその2　記憶容量が小さい！

別にディスクの容量や速度が毎日変化するわけではありません．物理的な速度や容量はメリットの項に書いた通りなのですが，それでも容量や速度を不満に思う時があります．その時はディスクの便利さに慣れきった時ですから，しばらくディスクの使用をやめて，カセットを使ってみることを勧めます．実際にディスクを使

い慣れてくると不便さが気になってくるのです．他にもディスクを趣味で使用している時には問題がなくても，業務用への使用を考えると少々物足りないかもしれませんが，これはプログラムの改善で使用できると思います．つまり，ディスクが遅いとか小容量というディメリットは，ディスクに慣れ切って人間が怠惰になった時か，本来ミニフロッピーディスクの性能では不足の分野でディスクシステムを使用した時に生じるのです．

ディメリットその3
複雑なソフトウエアが必要！

PC-8001のBASIC言語は，マイクロソフト社制作のN-BASICですが，PC-8031（8032）を接続するとN-BASICに種々のディスク用の機能が付加されて，Disk BASICに早変りします．Disk BASIC用に付加された機能は2章で詳しく説明しますがとりあえず2章のコマンド・ステートメント一覧表を見てください．どうです，数が多いでしょう．コマンドやステートメントの数が多いということは，使いやすいとも言えるのですが，使いこなすのが大変とも考えられます．

さらに最初に使うディスケットはフォーマットしなければならないとか，ディスクを使用する前に使用宣言，終了時には終了宣言をしなければなりません．終了宣言を忘れると，せっかく記録したデータを壊してしまいます．プログラム中でディスクを使用する時にも，使用宣言と終了宣言が必要です．

使用宣言や終了宣言の具体的な方法は第2章に，その動作の詳しい説明は第7章に書きますが，ここではなぜ使用宣言や終了宣言のような，〝おまじない〟が必要なのかを簡単に説明しましょう．

ディスクは早くて大容量という2つのメリットをランダムアクセス方式というアクセス方で実現していることは前に書きましたが，ランダムアクセス方式を行うには目的とするデータがディスケット上のどこに収まっているのか知っていなければなりません．シーケンシャルアクセス方式ではデータを順番に調べながら目的のデータを探せばよいのですが，ランダムアクセス方式でそんなことをすれば，せっかくのランダムアクセス方式のメリットが無意味になってしまうのです．そこでディスケット上のどこに何のデータが記録されているのかを記録した住所録のようなものを作り，それをディスケット上の一定の場所に記録しておきます．この住所録はディスケット上のデータを読む時には，データの住所（記録されている場所）を知るために参照されますし，データを書きこむ時にはどこが空地（記録されていない場所）かを知るために参照されると同時に，書き込んだ後には変更されなければなりません．つまり住所録は最もひんぱんに参照される最も重要な情報で，これが破壊されればディスケット上のデータは全てアクセス不能になるのです．話を前にもどして，ディスクの使用宣言と終了宣言ですが，これはPC-8001がPC-8031（8032）に新しく挿入されたディスケットの住所録を知る動作と，変更された住所録を再びディスケットへ書き込む動作の指定となるわけです．すなわち，使用宣言ではPC-8001は新しいディスケットの住所録を読み出します．それ以後はそのディスケットへのデータの読み出しや書き込みは，その住所録を参照しながら行うわけです．そして終了宣言では，ディスケットへの読み出しや書き込みを行った後の新しい住所録をディスケットへ書き込むのです．

ディスクシステムはソフトウェアが複雑になり，利用するのにもある程度の注意が必要という理由も，ディスクに対するアクセスの複雑さから来るものなのですが，高速で大容量というメリットを確保するために必要な処理ですし，実際にはほとんどの処理はDisk BASICが行い

ますから，使用する時にはマニュアルやこの本をよく読んで，基本的な注意を忘れないようにしてください．

ディメリットその4 弱い

さて〝弱い〟とは一体何が弱いのでしょう．答えは2つあります．

1)ハードウェア的に弱い――これはディスケット本体が物理的に弱いということです．ディスケット本体はソノシートより薄い円板で，不織布と厚紙でカバーされているだけですから，保存に注意しなければなりません．当然のことですが，曲げたり記録面に手で触れることも避けなければなりません．また磁石等を近づけるとせっかく記録したデータが破壊されてしまいます．

ディスケットはできるだけ専用の本立てのようなものに保管すればよいでしょう．

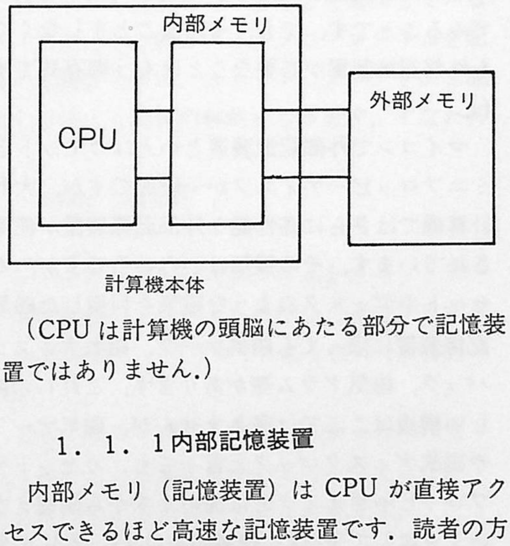
2)ソフトウェア的に弱い――これは先に書いたディスケットの住所録が破壊された時におこり，ディスク中のデータ全部を失ってしまいます．ディスクの使用宣言と終了宣言の指定を誤った時に住所録が破壊されます．住所録が壊れると記録されているディスクへのアクセスが不可能になりますから，もうそのディスケットは使いものになりません．ただ，ソフトウェア的に壊れたディスケットは，記録されているプログラムやデータがアクセスできないだけですから，ディスケットをもう1度フォーマットし直せば，プログラムやデータは消え去りますが，ディスケットは使用できます．

1 ディスクシステムの基礎

第０章ではディスクがどんな働きをするか，どんなメリットやディメリットを持っているかということを考えましたが，まとめると，ディスクはカセットテープよりも速くて大容量なデータの記憶装置であるということでした．さて，データの記憶装置といえばカセットやディスクだけではありません．大型計算機ともなればさらに高性能な記憶装置を用いています．第１章では，最初に計算機の記憶装置について説明して，フロッピーディスクが記憶装置としてどんな位置にあるかを知ってもらいます．それからPC-8031 (8032) がPC-8001の記憶装置として，データの記録をどのように行うかを説明したいと思います．

１．１　計算機の記憶装置の構成

計算機の記憶装置の構成を簡単に図で示しておきます．

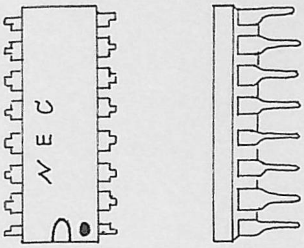


（CPUは計算機の頭脳にあたる部分で記憶装置ではありません.）

１．１．１内部記憶装置

内部メモリ（記憶装置）はCPUが直接アクセスできるほど高速な記憶装置です．読者の方も今までにRAMという言葉を一度は聴いたことがあると思いますが，RAMもROMも計算機の内部記憶装置として使用されるメモリです．PC-8001のRAMは図のような小型ＬＳＩメモリを，16Kシステムで８個，32Kシステムで16

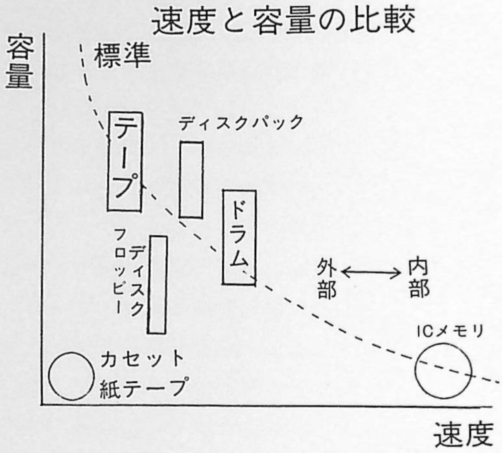
個使用しています．



1．1．2外部記憶装置

もう一度計算機の記憶装置の構成図を見て下さい．外部メモリ（外部記憶装置）は計算機本体の枠の外に描かれていますね．これが外部記憶装置と言う理由です．ディスクやカセットも外部記憶装置のうちの一つですが，外部記憶装置についての説明にはいる前に，なぜ外部記憶装置が必要なのか考えてみて下さい．それにはためしに1ヶ月間カセットやディスクなど一切の外部記憶装置を使用しないでPC-8001を使ってみることです．でも，そんなことをしなくても外部記憶装置が必要なことはもう御存知ですね．

マイコンで外部記憶装置といえばカセットとミニフロッピーディスクが一般的ですが，大形計算機ではさらに高性能な外部記憶装置が使用されています．その種類はいろいろですが，カセットやディスクのような磁気を利用した磁気記憶装置に限っても磁気テープ，磁気ディスクパック，磁気ドラム等があります．これらの詳しい構造はここでは書きませんが，磁気テープや磁気ディスクパックと言っても，カセットやフロッピーディスクとは違いますから間違えないようにして下さい．読者の方はPC-8031の価格を安いと思いますか？　大抵の人は高いと思うでしょう．しかしそれでも磁気テープ，磁気ドラム，磁気ディスクパックに比較すると割安なのです．



容量と速度の関係のグラフを見て下さい．高速で小容量な記憶装置としてICメモリがあり，低速で大容量な記憶装置として磁気テープがあることがわかると思います．このグラフ中で価格はどう扱われているかというと，先に書いたように高速で大容量になるに従って高価になりますから，グラフの原点から離れるほど高価になるわけです．その意味でフロッピーディスクシステムは，磁気テープ，磁気ディスクパック，磁気ドラムほど容量もスピードもありませんが価格が安いので我々アマチュアとかスモールビジネス用に使用できるわけです．

実際のところマイコンにフロッピーディスク以上の高性能な外部記憶装置をつけてみてもマイコンの性能速度の点で効果的ではないことが多いと思います．

＊＊　もちろんマイコンの外部記憶装置としてはフロッピーディスクが一番良いと決定されたわけではありません．選択にはコンピュータの性能や，外部記憶装置の性能の他に，システムを使う目的も関係してきます．たとえばスモールビジネスや科学技術用のデータ処理では，フロッピーディスクでは容量や速度の点で不足になることがあるでしょう．そういう時はもっと高性能な外部記憶装置を考てみる必要があるかもしれません．　＊＊

1．2　ディスク上のデータ扱い

ディスクはデータを記録するものですが，そのデータを人間が扱うにはどうすればよいでしょうか．例えばカセットにプログラムを保管する時のことを考えてみましょう．

プログラム〝ASCII〞をカセットにセーブするには

csave 〝ASCII〞

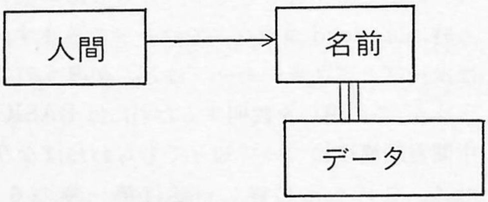
と打ちこみました．また，そのプログラムをロードするには

cload 〝ASCII〞

と打ちこみました．

この動作の中で我々人間は，プログラムというデータをASCIIという名前で扱っていたわけです．ディスクの場合も同様で記録するデータは名前で扱うことができます．

データに名前をつけるということを図に書けば



という扱いをしていることになりますが，ここでデータに名前をつけて扱うのではなくデータを直接指す概念として，ファイルという言葉を説明します．

ファイルとは「意味をもったデータの集合体」と言えます．別にコンピュータ用語に限らず身近なもので考えてみてください．例えばノートをまとめてファイルすると言いますが，ノートは私達にとって意味をもったデータですからそれを整理したものはファイルと言えるわけです．要は人間がデータを扱う時の概念としてファイルという言葉を用いることを知っておいて下さい．

ディスク上にデータを記録するには，ファイル単位で記録します．例えば一本のプログラムをディスク上に記録するということは，ディスク上に一つのファイルを作ることになるわけです．先にディスク上にデータを記録する時にはデータに名前をつけて取り扱うことを書きましたが，名前をつけるということは，ファイルに名前をつけるということで，この名前をファイル名といいます．

実際の動作で説明しましょう．ディスクにプログラムを記録するコマンドは save ですが

save 〝ASCII〞

とキーを打つことは，プログラムをディスクに記録してファイルを作る，そのファイルにはASCIIというファイル名をつけることを意味しているわけです．上の例はディスクにプログラムを記録する場合ですが，カセットにプログラムを記録する時にも同様の解釈ができます．すなわち

csave 〝ASCII〞

とキーを打つことはプログラムをカセットテープ上に記録してファイルを作り，そのファイルには ASCII というファイル名を付けているわけです．

カセットテープしか使用しなかった時にもファイルという言葉こそ用いませんでしたが，ファイルの考え方は用いていたわけです．

ファイルの概念を理解できたでしょうか？　ファイルはデータの論理的な取り扱いの単位のようなものですから，無理に一つの定義をあてはめて考えないで言葉に慣れるようにして下さい．この本を読み進むうちにファイルという言葉は

山ほど出てきますから，自然と慣れると思います．この本ではここからは，ディスクはファイルを保管するものとして考えます．

1．3　ファイルの形式

コンピュータがディスクにデータを書き込んだり，ディスクからデータを読み出したりすることは先に書きました．そしてそのデータをファイルと呼ぶことも書きました．さて次に書くことは，ファイルの中味についてです．

ここまでの例ではディスクに制作したファイルはプログラムだけでした．しかし，カセットのことを思い出して下さい．カセット上に作られたファイルはプログラムだけだったでしょうか？　プログラムの他にデータも記録できましたね．

PRINT　#-1

を使ってカセット上にデータを記録したことを思い出して下さい．プログラムではないデータをカセット上に記録してもそのデータはファイルと呼べますから，このことはカセット上にデータのファイルを作ったことになります．ディスクにプログラム以外のデータは記録できるでしょうか？　当然できます．非力なカセットでもできるのです．ディスクができないはずはありません．

ディスク上へのデータの書込みの具体的な話は第2章でします．とりあえずここまでの話で，ファイルにはプログラムを書込んだものと，データを書込んだものの，2種類あることが理解できたでしょう．プログラムを書込んだファイルをプログラムファイル，データを書込んだファイルをデータファイルと呼びます．ファイルはプログラムファイルとデータファイルをまとめた呼び方なわけです．

ファイル｛プログラムファイル
　　　　　データファイル

1．3．1　プログラムファイル

プログラムファイルとは，プログラムが保管されているファイルですから作るのは簡単です．ディスクへプログラムを記録するためには，

save

という便利なコマンドがありますから，ASCII というプログラムファイルを作るには

save "ASCII"

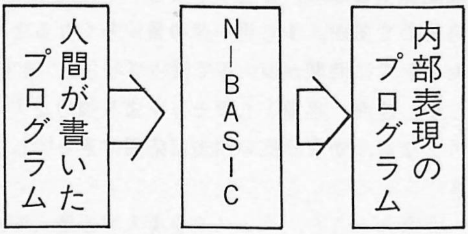
でいいわけですが，もう1つ，タイプの異なる save コマンドがあります．

save "ASCII" ,a

がそれです．コマンドの最後に ,a をつけただけです．,a をつけない save をセーブコマンドと言い，つけたコマンドをアスキーセーブコマンドと言います．セーブとアスキーセーブはどちらも外見的な機能は同じで，プログラムをディスクにセーブすることです．もちろんロードする時には，load コマンドでロードできます．ではセーブとアスキーセーブはどこが違うのでしょうか．この違いを説明するためには BASIC の中間言語構造について知ってもらわねばなりません．このへんの詳しい話は第一巻の 6・2 BASIC のキーワードの項（121ページ）に書いてありますが，ここでも簡単に説明します．

プログラムは最初は人間の手でキーボードから入力されるわけですが，N-BASICの文法を守って入力されたプログラムは即座に N-BASIC が解読して，内部表現に変換してしまいます．従ってメモリ内のプログラム格納領域に格納されているプログラムは人間が手で入力した形とは異なっているわけです．

その様子を図示すると



のようになるのでしょうか.

BASICがわざわざプログラム形を変えてしまうのは

1) 人間が書いたプログラムを短い形に変換してメモリに格納してしまい，メモリを効率良く使用する.

2) プログラムをコンピュータが解釈しやすい形に変換してしまい，実行する時の速度を向上する.

という2つの理由があるのですが，逆に一度メモリに納めてしまったプログラムを見直す時には，内部表現を人間が書いたプログラムに逆変換しなければなりません．例えば

list

と打つことは，内部表現形式のプログラムを人間が書いた表現にして表示することになるわけです.

とりあえずBASICのプログラムには表現上で2種類あることを理解して下さい.

さて話をまたセーブと ,a をつけたセーブに戻します．2つのセーブは外見的な機能は同じということは先に書きましたが，もう少しヒントをあげますから違いを考えてください.

2つのセーブはプログラムを異ったタイプでセーブするのです.

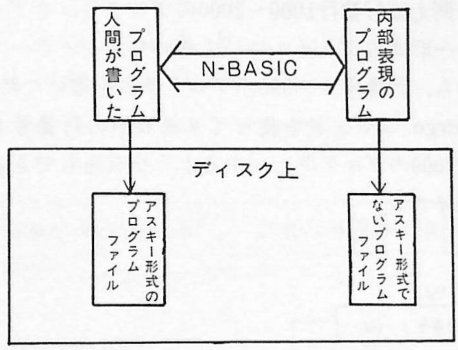
どうです？ だいたい察しがついたでしょう。

save"ASCII"

のコマンドではASCIIというプログラムは一度BASICで変換された内部表現の形でディスクに記録されます。

save "ASCII" ,a

のコマンドではASCIIというプログラムが，人間が書いたプログラムの形でディスクに記録されます．人間が書いたプログラムはコンピュータではアスキーコードの列として受け取られますから，この ,a をつけてセーブしたプログラムファイルはアスキー形式のプログラムファイルと呼ばれます.



先にファイルには，プログラムファイルとデータファイルがあることを書きました．そしてプログラムファイルはアスキー形式とアスキー形式ではないファイルがあることを書きました．ここまでの段階でファイルを分類すると

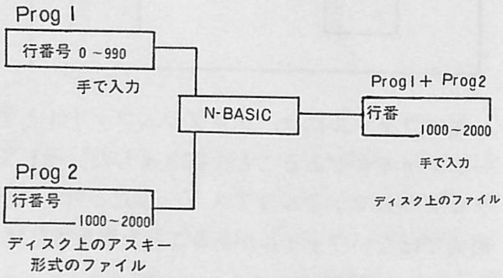
プログラムファイル { アスキー形式
中間言語形式

と書けます.

さて，プログラムファイルは内容がプログラムですから，ロードして実行できる形で記録されているなら，記録されてる形は問題ではないはずです．先にアスキー形式以外のプログラム（内部表現に変換されたプログラム）は，アスキー形式のプログラムより短く変換されていることを書きました．この特長はディスク上に記録されても受け継がれますから，同一プログラムをファイルした場合，アスキー形式のプログ

ラムファイルは内部表現形式のプログラムファイルより長くなります．従って，ディスクの効果的活用という意味では内部表現形式のファイルのほうが優れているわけです．しかしアスキー形式のファイルは人間がキーボードから入力したのと同じ形をしていますから，ディスク上のアスキー形式のファイルを読む時，キーボードから入力することと同じように扱うことができます．

例えば行番行1000～2000のプログラムをアスキー形式のファイルにしておいて，キーボードから，行番号0～990のプログラムを書いた時，merge コマンドを使ってメモリ中の行番号0～2000のプログラムにするような芸当もできるわけです．



1．3．2　データファイル

PC-8001の外部記憶装置にカセットしか使用しなかった時にも，データファイルをカセットテープ上に作っていたわけですが，読者の方で，PC-8001を持っておられる方のうち一体何人ぐらいが，カセットを使ったデータファイルを使ってみたでしょうか．PC #1の7・5　実践!!　プログラムの開発の項（140ページ）で書いた名簿管理プログラムは，カセット上にデータファイルを作っているよい例ですが，実際にあのプログラムを動かしてみた人なら知っているでしょうが，大きな欠点があります．データをカセットへ書く時とカセットから読む時に長い長い時間がかかるのです．変数やストリング変数を2～3個記録するなら，カセットでもアッと言う間に終るのですが，少しデータの量が多くなるとカセットでは時間がかかって使いづらくなります．しかし速度，容量ともカセットより強力なディスクなら，かなり広い用途に応用できるでしょう．

実際のところ，ディスクを導入する最大のメリットは，データの扱いが強力になるということなのです．読者の方も，ディスクを購入した暁には，プログラムファイルのロードやセーブが高速にできることを喜ぶだけではなく，ディスクのデータ処理能力を生かすプログラムに挑戦して下さい．それがディスクシステムを本当に活用することなのですから．

さて，データファイルの説明に入りましょう．プログラムファイルは，その記録方式の違いで2つのタイプに分類されました．

プログラムファイル｛アスキー形式
　　　　　　　　　　中間言語形式

上のように，アスキータイプと内部表現タイプでしたね．プログラムファイルはデータの記録形式で二つに分類しましたが，データファイルは異なった意味で2つのタイプに分類されます．シーケンシャルファイルとランダムファイルです．

シーケンシャルとランダムという言葉を覚えていますか？　第0章のディスクのメリットその3で，シーケンシャルアクセスとランダムアクセスについて説明しました．シーケンシャルアクセスとランダムアクセスは，記憶装置のアクセス方式でした．もし忘れているなら，もう一度第0章のディスクのメリットその3を読み直して下さい．このアクセス法の違いが，そのまま2つのデータファイルの型の違いに通じていますから．

さて，シーケンシャルファイルと，ランダム

ファイルについて詳しく説明する前に，データファイルにデータを読み書きする単位として，レコードという言葉が用いられますから，レコードについて説明します．

レコードの定義は次のようになります．

「レコードとは，一度にファイルに入出力されるデータの組である．」

レコードと，ファイルの関係について注意して下さい．レコードは，ファイル内のデータで，入出力が1回で行われるものです．

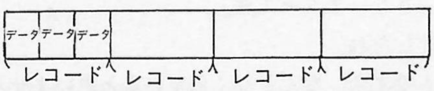
従って

ファイルはレコードの集合体

レコードはデータの集合体

となるわけです．

その様子を図示すると，

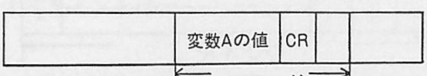


のようになるでしょう．

具体的に，カセットにデータファイルを作る時のことを考えてみましょう．

print #-1, A

とキーボードから打つことは，カセットテープに変数Aの値を記録することですが，テープ上には

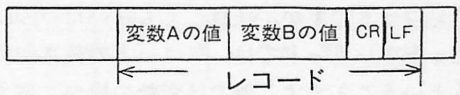


の形で記録されています．変数Aの値は1度にファイルに書込まれますから，これをレコードと言うわけです．CR／LFはキャリッジリターンラインフィードのコードで，レコードの区切りに使われています．

変数が2個の場合は

print #-1, A, B

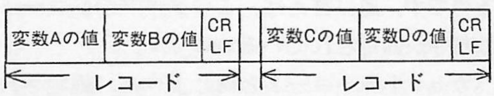
ですが，この時テープ上の記録の形は



となります．CR／LFは変数A，Bの次に記録されていますから，変数A，Bの値が組になって1つのレコードを構成することがわかるでしょう．ついでにもう1つ

print #-1, A, B
print #-1, C, D

を実行した時のテープ上の形は



となります．レコードが2つになりましたね．

3つの例で，カセットの場合はレコードがどのように作られるかわかると思います．

まず

print #-1, A, B

では，カセットに，変数Aの値とBの値を1度に書込みます．したがって，2つの値の組で1つのレコードとなるわけです．

次に

print #-1, A, B
print #-1, C, D

では，最初に変数Aの値とBの値の組を1度に書込んで，次に変数Cの値とDの値の組を1度に書込まれるわけです．書込み動作は2回ですから2つのレコードとなります．

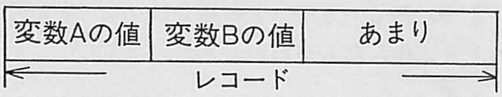
要は一回の　PRINT#-1　で書込むデータの

組がレコードとなるわけです.

これがカセット上にデータファイルを使う場合のレコードですが，注目してもらいたいのはカセットのレコードでは，レコードの長さが変わるということです．例では変数の組が２個までしたが，３個でも，４個でも組にすることができますし，255個の文字列を値に持つ文字変数かもしれません．とにかくデータの組が大きくなれば，レコードの長さも長くなるわけです.

カセットにデータファイルを作る時の，レコードの意味を理解できたでしょうか．レコードの長さが変化することに注目してほしいと書いたのは理由がありまして，実はもう１つ，レコードの長さの変化しないタイプのレコードがあるのです．逆に言えば，このタイプのレコードは長さが固定されているわけです.

カセットのレコードと同じように，固定された長さのレコードに，変数 A と B の値を記録した様子を図示すると，



という具合でしょう．この場合はカセットの例のように，レコードの終りを示すCR／LFのコードは必要ありませんね.

レコードの長さが固定されている場合，レコードに記録するデータの量がレコードの長さに満たないと，レコード内に余りができます．従ってファイルの長さの割には，データの量は少いということになります.

しかし，この固定長のレコード方式にも捨て難い魅力があります．このことは，この後シーケンシャルファイルとデータファイルの説明をする時に詳しく書きますが，とにかく今は可変長のレコードと固定長のレコードがあり，カセット上のファイルを使用する時には，可変長のレコード方式が使用されていることを覚えておいて下さい.

1．3．2．1 シーケンシャルファイル

データファイルは，シーケンシャルファイルとランダムファイルに分類されると先に書きました．まずシーケンシャルファイルについて説明しましょう.

シーケンシャルファイルを，簡単に説明すると，「書いた順序でしか読めないファイル」と言うことができます.

上の説明の「書いた順序でしか読めない」の部分に注目して下さい．こんな説明をどこかで読んだと思いませんか？　実は第０章の，シーケンシャルアクセス方式の説明の所で，これと似たような説明をしたのです.

シーケンシャルアクセス方式は

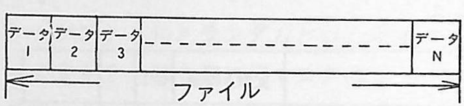
「特定の順序でしか，データをアクセスできないアクセス法」

でしたね.

どうです？　なんとなく似ていると思いませんか．似ているのも当然で，実はシーケンシャルファイルという呼名は，ファイル内のデータをアクセスする方法が，シーケンシャルアクセス方式である点に由来しているのです.

シーケンシャルファイルを，もう少し詳しく説明しましょう.

ファイル中にデータが記録されている様子を図示すると，下の様になります.



この図は，N 個のデータで構成されたファイルを示しています.

さて，図のようなファイルからデータを読む時のことを考えてみましょう．シーケンシャルファイルでは，データは書込んだ順序でしか読めませんから，

１回目の読み取りでデータ１の値が読める.

2回目の読み取りでデータ2の値が読める.

N 回目の読み取りでデータ N の値が読める.

N+1回目の読み取りでは,データがないのでもう読めない.

ということになります. これ以外の読み取りはできません. 従って, N 個目のデータだけ読みたい場合でも, N-1個のデータを読みとばす（空読みする）必要があるわけです.

＊＊＊シーケンシャルファイルの場合, データをアクセスする方法は, シーケンシャルアクセス方式であるということが理解できたでしょう. ＊＊＊

もう少し具体的に話しましょう. 具体的にといってもディスクの話ではなく, カセットの話です. シーケンシャルファイルをディスクで扱う具体的な説明は第2章でします. ここではできるだけシーケンシャルファイルについて正確に理解するようにして下さい.

ここまでシーケンシャルファイルの説明を読んだ人は, カセット上のファイルはシーケンシャルファイルであるということを, すぐ理解できるでしょう. カセットにファイルを作る時のステートメントは

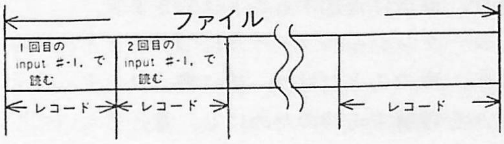
print #-1, 〈変数の組〉

で, 読む時のステートメントは

input #-1, 〈変数の組〉

でしたね.

print #-1, を実行する時の, カセットテープへ記録する様子はレコードの説明の所で書いたので, 今度は input #-1, の様子を図にしてみましょう.



こんな具合です. 1回の input #-1, で読むデータの組を, レコードと言うことはわかるでしょう. 上の図でもわかるように, 1回目の input #-1, で最初のレコード(指定された数のデータの組) を読み, 次の input #-1, で2番目のレコードを読む, といった具合に読みますから, テープに記録された時の順序に従わないで, データを読むことはできません. ですから, カセット上のファイルはシーケンシャルファイルとなるわけです.

次に, カセット上のファイルに対する制限を考えてみましょう. もちろんシーケンシャルファイルですから,「書いた順序でしか読めない」という制限はあります. それ以外です. それは一度記録したデータは, 書き換えることができないということです. 一度作ったら, データはもちろん, レコード単位でもテープ上で書き直すことはできません. 少しでも修正したければファイルを読んで修正しながら, 新しいファイルに書き込む手順が必要です. 要は, 一部のレコードだけの書換えはできないわけです.

ここまで, カセット上にファイルを作る時のことを例にして, シーケンシャルファイルについて説明しましたが, 実はディスク上にシーケンシャルファイルを作る時も, ファイルの構造はカセット上に作ったファイルと全く同じなのです. また, ファイルに対する制限もほとんど同じです. ですからファイルを読み書きするときには, カセット上のファイルの動作の説明を, もう一度読んでくれればいいわけです. 制限も「ファイルの一部を修正することはできない」ということです. ただ, ディスク上のシーケンシャルファイルの場合は, 一部の変更は無理で

すが，最後に追加することはできます．

シーケンシャルファイルについてまとめてみます．次のことだけは，後に書くランダムファイルを理解する時のためにも，覚えておいて下さい．

*シーケンシャルファイルとは，書いた順序でしか読めないファイルである．

*ファイルの一部を変更することはできない．できるのは，最後へ追加することだけである．

*一度に読み書きできるデータ（レコード）の長さは変えることができる．

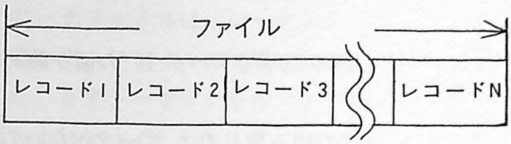
1．3．2．2　ランダムファイル

シーケンシャルファイルの説明を読んだ方は，ランダムファイルがどんなものであるか，なんとなく想像がついたと思います。そしてたぶんその想像はあたっていることでしょう。ランダムファイルとは，すなわち「任意の場所をアクセスできるファイル」とでも言えるものなのです。簡単にシーケンシャルファイルと比較しながら説明するとランダムファイルとは次のようになります。

1) シーケンシャルファイルに書かれたデータを読む時は，書いた順序でしか読めませんでしたが，ランダムファイルでは，何番目のレコードでも自由に読むことができます．

2) シーケンシャルファイルでは，一部のレコードの書換えはできませんでしたが，ランダムファイルでは，何番目のレコードでも，自由にレコード単位の書換えができます．

どうです．シーケンシャルファイルと比較すると，ランダムファイルの特長がよくわかるのではありませんか．さて，もっと詳しく，下図のモデルを使って説明しましょう．



さて，上の図はN個のレコードで構成される，ランダムファイルを示しています．このランダムファイルに対して実行できることは，

1) レコード番号n（1≦n≦N）のレコードを読み出す．

2) レコード番号n（1≦n≦N）のレコードに書き込む．

3) レコード番号がN以上のレコードへ書き込む．

の3点です．

1)と2)は，先に書いたランダムファイルの2つの特長「何番目のレコードでも読める」と「何番目のレコードでも書換えできる」を書きなおしたものです．

3)はファイルへ追加する機能を示しています．

ランダムファイルがどんなものか理解できたでしょうか．全然わからない人には，ランダムファイルの説明をもう一度くわしく説明いたします，なんとなく分ったような気のする人は，どうぞ先へ読み進んでもらっても結構です．

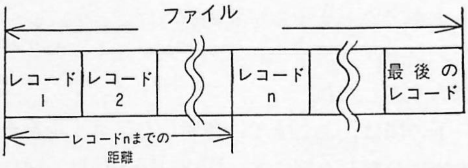
ランダムファイルの特長は，何番目のレコードでもアクセスできることですが，なぜこんなことができるのかを説明します．これは，逆に言えば，なぜシーケンシャルファイルでは任意のレコードをアクセスできないかに通じていることです．

任意のレコードをアクセスするためには，2つの条件が必要です．

・「n番目のレコード」をアクセスしようとするとき，その「n」の数字から，レコードがファイル中のどこにあるかを，知ることができなければなりません．

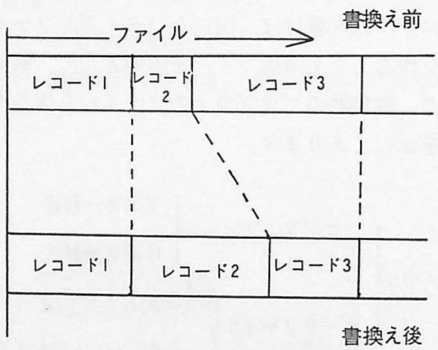
·レコードを書き換える時のことを考えると，書き換えた後のレコードの長さが，書き換える前のレコードの長さより長くなってはいけません．

さて，最初の条件の意味は分るでしょう．「n番目のレコードの場所」というのは，ファイルの先頭から，n番目のレコードまでの距離とも言えます．その様子を図に描くと，



こんな具合です．

2番の条件を考えます．悪い例を図示すれば，すぐ理解できると思います．2番目のレコードを書換える時の，書換える前後の図です．



レコード2の長さが，書換え前より長くなったら，レコード3の長さが短くなってしまいました．実はレコード3は短くなったのではなく，破壊されてしまったのです．正確に言えば，

「レコード2が長くなってしまい，本来ならレコード3の記録部分にまで及び，レコード3の一部を消してしまった.」

ということです．これを防ぐ方法がないわけではありません．レコード2が延長されるならば，レコード3を後方にずらせばよいのです．レコード3を後方にズラすためには，レコード4を後方にズラして，さらにレコード4を後方にずらすためには，レコード5を後方へずらして………．結局書換えるレコード以後を，全部後方にズラさなければならないわけです．これは手間のかかる操作ですし，この操作をすると，先に書いた「ファイル中のレコードの場所」が書換えをするたびに変化しますから，いよいよランダムにアクセスできるファイルを構成することは難しくなります．

ランダムファイルというのは，任意のレコードの書換えができるという，非常に便利な特長を持っていますが，そのために2つの条件を満足するような構造のファイルを実現しなければならないわけです．したがって，ランダムファイルを作る手順は，シーケンシャルファイルを作る手順よりも少し複雑になります．実際の具体的な手順は第2章と第3章で書くことにして，ここではどの様に2つの条件を満足するファイルを構成しているかを説明しましょう．

まず結論を先に書けば，レコードの長さを固定してそれを実現しているのです．固定長のレコードを覚えていますか？ レコードの説明の最後に，少し説明しましたが，忘れているなら，もう1度読み直して下さい．

固定長のレコードを採用することでランダムファイルを実現するための2つの条件は，解決されます．第2の条件の

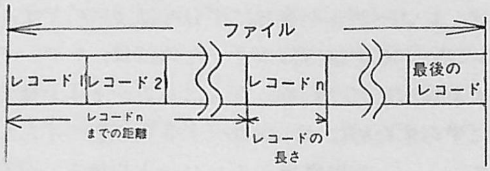
「レコードを書換える前の長さが，書換え後の長さより短くてはいけない.」

は，考えるまでもなく満たされています．

第1の条件の

「レコードが，ファイル中のどこにあるかを知ることができなければならない.」

が満たされているということも，少し考えれば理解できます．これはファイルの先頭からの距離が分ればよいのですから．



n 番目のレコードまでの距離は,

レコードの長さ＊ (n-1)

の簡単な式で示されることを使えばよいのです. アクセスしたいレコード番号を与えれば, この式を使って, 即座にレコードの存在する場所が分ります.

どうです？　任意のレコードをアクセスできるファイルの構造が理解できましたか？　要するに, 固定長のレコード方式を用いることによって, すべての問題を解決しているわけです. 本当のところ, 固定長レコード方式は, ディスクのハードウェアにとっても都合がいいのです. この話はこの後, 使用宣言と終了宣言の項でも簡単に書きます. 詳しい話は第6章と第7章を読んで下さい.

最後に, ランダムファイルのこれまで説明できなかった特長を書いておきます.

・まず最初にあげられるのが, 任意のレコードをアクセスするのに, 時間がかからないということです. シーケンシャルファイルでは任意のレコードを読むためには, レコードを一個一個読まなければなりません. ランダムファイルでは, 途中のレコードを省いてアクセスできますから当然速度は速いはずです.

・ランダムファイルでは, ファイルの使用効率が悪くなります. 固定長のレコードは, レコード内がデータで埋まることはほとんどありません. 従って, ファイル全体の長さの割にはデータの量が少くなります.

1．3．3　ファイルの種類のまとめ

ファイルの種類と構造を説明してきましたが, ここで, これまで説明してきたファイルのいろいろな性質をもう一度復習してみましょう.

まず, ファイルはプログラムを保存するためのプログラムファイルと, データを保存するためのデータファイルに分類できました. そしてプログラムファイルには, 人間が入力した形のアスキー形式のファイルと, BASIC の中間言語になおした時の形の非アスキー形式のファイルがありました.

一方, データファイルには, ファイル中のデータをアクセスする方式の違いによって, シーケンシャルファイルとランダムファイルの2つがありましたね.

下の図は, ここまでに説明したファイルを系統的に分類した図です. Disk-BASIC は, 図に書かれた全てのファイルを操作する機能をもっているわけです. ただ, 誤解しないでほしいのですが, ファイルの種類は図に示したものだけではありません. 上に示した図は, Disk-BASIC が操作できるファイルの種類ですから, 自分でプログラムを書けば, 図にないタイプのファイルを作ることも可能です. その例として, 第4章で, 機械語のプログラムのファイルを作る方法を示してあります.

- ファイル
  - プログラムファイル
    - アスキー形式
    - 中間言語形式
  - データファイル
    - ランダムファイル
    - シーケンシャルファイル

1．4　使用宣言と終了宣言

1．4．1　基礎知識

第0章で軽く触れましたが, ディスクシステムを使用する時には, 使用宣言と終了宣言を欠かすことができません. しかも, この宣言にはディスケットの使用に対する宣言と, データファイルの使用に対する宣言があり, 事実上2重

に宣言することが必要です．なぜこんな宣言が必要なのかを知るには，次の３点についての知識が必要です．

1)ディスケット中のファイルの構造
2)ファイル中のデータの構造
3)ディスケット中のデータのアクセス方法

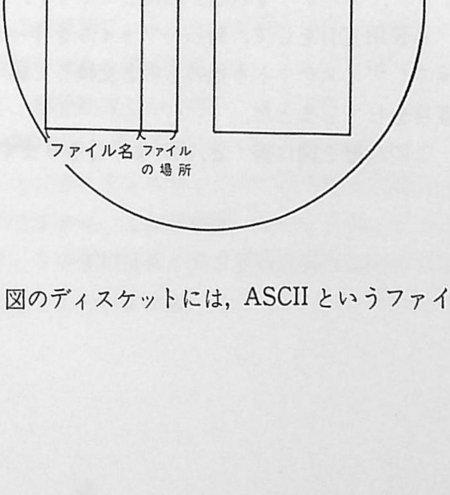
1)については，第０章で簡単に説明してあります．第０章では，「データがディスケット中のどこに記録されているかを示す．住所録のようなものがあって……」と書きましたが，実はこの説明の中のデータとは，ファイルのことなのです．従って，「データが……」の説明は

「ディスケット中には，ファイルと，ファイルがディスケット中のどこに記録されているかを示す配置表（住所録）が存在する」

と書き直せるわけです．

配置表には，ファイル名とファイルが記録されている場所が登録されています．

この様子を図示すると，次のようになるでしょう．



図のディスケットには，ASCII というファイルがすでに記録されていることもわかると思います．

図のように，名前と場所を記録した配置表を用いると，人間がファイルを扱う時には，全てファイル名で扱うことができるのです．ファイル名を指定すれば，即ファイルの場所が分りますから，人間様がわざわざファイルの場所を考える必要はないわけです．もっとも，実は配置表はこんなに単純なものではなく，もっと複雑な構造になっているのですが，使用宣言と終了宣言を理解するための知識としては，これで十分です．もっと詳しく知りたい人は第７章を読んで下さい．

2)のファイル中のデータの構成は，前節の１・３ファイルの種類で書きました．ここを読めば十分です．

3)については注意が必要です．これはディスケットにデータをどんな構成で記録するかというソフトウェア的問題ではなく，記録するためのハードウェア的問題のことだからです．ちょっとややこしいのですが，ここでは次のことだけを覚えておいて下さい．

＊ディスクへのデータの読み書きは，256バイト単位でしか行えない．

256バイトというのは，純粋にハードウェア上の規定で，１バイトや２バイト単位では不可能なのです．

この事実を知ると，シーケンシャルファイルの構造の所で筆者が嘘を書いたと思われる読者がいるかもしれませんが，そうではありません．ファイルというのは，データの集合を論理的に考えた単位で、ディスケットに書き込む単位というのは，データの集合の物理的単位であるからです．

論理的単位と，物理的単位にはギャップがありますが，Disk　BASIC がそのギャップを埋めてくれますから，我々は論理的単位のファイルを扱っていればよいわけです．

とにかく，ここではディスクへ読み書きは，256バイト単位ということを覚えておいて下さい．

基礎的な知識を得たら，次は使用宣言と終了宣言の動作について説明します．前にも書いたように，使用宣言と終了宣言はディスケットに対するものと，データファイルに対するものがあります．まずディスケットに対する宣言を説明して，次にデータファイルに対する宣言を説明しましょう．

1．4．2　ディスケットに対する宣言

ディスケットの使用宣言は mount というコマンドで行ない終了宣言は remove コマンドで行ないます．実際の使い方は第2章を読んで下さい．ここで説明するのは2つのコマンドの動作となぜ必要なのかという理由です．

前にディスケット中のファイルをアクセスするために、ディスケット中にはファイルの住所録が記録されていることを書きましたが，使用宣言と終了宣言は，この住所録に対して実行するコマンドなのです．もう少し具体的にどうやって宣言を行なうか考えてみましょう．

1)まずドライブにディスケットを挿入します．

2)ディスケットの使用宣言をします．すると、ディスケット中の住所録が，PC-8001の内部にコピーされます．

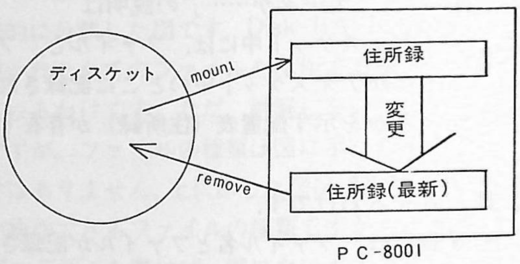
3)ディスケット中のファイルを読む，又は新しいファイルを作る等，ファイルの操作を行ないます．この時住所録を書き直しますが，この操作は PC-8001にコピーされた住所録に対して行われます．

4)ディスケットの終了宣言をします．終了宣言によって PC-8001 内部にコピーされていた住所録が，デイスケットに書き込まれます．

5)ドライブからディスケットを取り出します．

この手順を読めば分るでしょうが，ファイルを操作する時には，PC-8001内部にコピーされている住所録を使うわけです．住所録の変更もコピーされているものに対して行われますから，最新の住所録は PC-8001の内部にある住所録で，ディスケット中の住所録ではありません．最新の住所録は終了宣言でディスケットに書込まれるわけです．

ディスケットの住所録の正常な動きは下図のようになります．

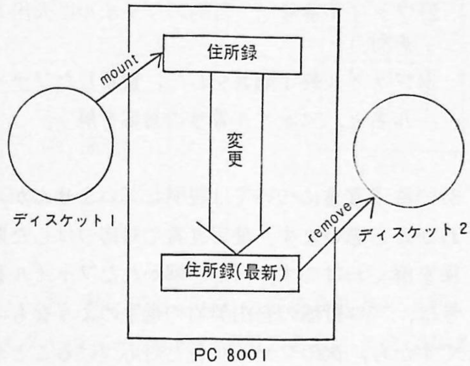


住所録は1枚のディスケット中のファイルの配置ですから，個々のディスケットで内容が違いますし，一枚のディスケットでも，ファイルの操作を行うたびに変化します．ディスケットの使用宣言と終了宣言は，変化の激しい住所録を最新に保つために必要な処理なわけです．

さて，このディスケットに対する使用宣言と終了宣言が確実に行われないと，どうなるでしょうか．次のような状態を想像して下さい．

＊使用宣言をして，新しいファイルを作った後で，ディスケットを他のものと交換して終了宣言をしてしまった．

この状態を図に書くと，次のようになります．



まず，使用宣言及び新ファイルを作ったディスケットについて考えてみましょう．

新ファイルを作ったので，ディスケット中には新ファイルが記録されます．又，PC 内部の住所録にも新ファイルが登録されます．しかしこのディスケットに対して終了宣言を行わなかったので，新しい住所録は記録されず，このディスケット中の住所録には新ファイルは記入されていません．

次に，終了宣言を行ったディスケットについて考えてみましょう．

このディスケットへは，ファイルの書込みはしていませんが，ディスケット1の新しい住所録が書き込まれています．

結局，2枚のディスケットの両方とも，住所録が実際のファイルの名前や位置とかみ合わなくなったわけです．

この状態は，あなたが見知らぬ町で間違った地図を頼りに郵便配達をするような状態なのです．郵便配達ならば聞くこともできますが，人に聞けない PC-8001 は，ただオロオロ….

とにかくディスケット中の住所録は大切なものですから，これを破壊しないように，ディスケットの使用宣言と終了宣言は忘れないようにしないといけません．

1．4．3　データファイルに対する宣言

データファイルに対する使用宣言と終了宣言は，プログラム中でデータファイルを使用することの宣言と，使用を終えたことの宣言で，open とclose で宣言します．実際の使用に当っては，使用宣言のモードとか使い方があるわけですが，その話は第2章でします．ここで説明するのは，宣言がどんな動作をするかであり，何故必要なのかという理由です．

さて使用宣言によって PC-8001がどんな動作をするのか説明しましょう．データファイルの使用宣言によって，次の2つの動作が行われます．

＊ファイルをプログラム中でどう呼ぶか決める．

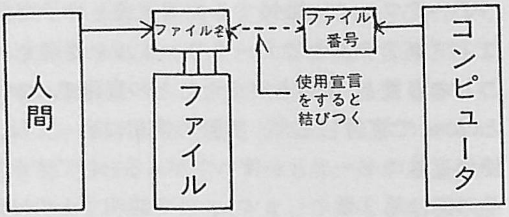
＊ディスケットへ書き込んだり，ディスケットから読み出すための領域を割当てる．

何やら理解しがたいことのように思われる方もいらっしゃるでしょうが，実際に使用する時に理解していなければならないのは，最初に書いたプログラム中での呼び方だけです．従って興味のない方は，2番目の説明は読まなくてもかまいません．逆に，更に詳しく知りたいと思った方は，第6章や第7章を読んで下さい．

実際の説明に入りましょう．まず最初の「プログラム中でファイルをどう呼ぶかを決める」についてです．

先に書いたように，我々人間はファイルを扱うときはファイル名で扱います．しかし Disk-BASIC では，データファイルは，ファイル名で入出力を行うことはできません．ファイル番号で入出力を行います．従って，あらかじめファイル名とファイル番号の対応を宣言する必要があるわけです．

このことを図示すると，下のようになります．



人間はファイルをファイル名で扱いますが，コンピュータはファイル番号で扱います．ファイル番号とファイル名を対応（図の破線部）させるのが，ファイル使用宣言です．プログラムのファイルの呼び名とは，このファイル番号を意味しているわけです．

具体的なファイル番号の形は，#1～#15のうちどれかですが，条件がつきます．

実際に Disk-BASIC を使った方は知っていると思いますが，システムをセットした時，Disk-BASIC は

How Many Files（0-15）?

と聞いています．この質問の意味は，後で説明しますが，ここでは，ファイル番号をいくつまで使うのか聞いていると考えてかまいません．

つまり，最初の質問に，〝7〞と答えれば，ファイル番号は，#1～#7までしか使用できないわけです．もし最初の質問に〝0〞と答えればプログラム中でデータファイルを使用できないことになります．

ファイル名と，ファイル番号の対応づけについて理解できたでしょうか．理解できれば，データファイルを使用するプログラムの手順もわかってくると思います．

プログラム中でファイルを使用するには次の手順を踏むわけです．

1)ファイルの使用宣言をして，使用するファイル名と，ファイル番号の対応づけをする．
2)ファイル番号で，目的のファイルに入出力を行う．
3)ファイル終了宣言をして，使用したファイル名と，ファイル番号の対応を解く．

3)の終了宣言については説明していませんが，わかると思います．使用宣言で対応づけした関係を解くわけです．対応を解かれたファイル番号は，プロ野球の自由契約の選手のようなものですから，次のファイル名と対応づけることができます．

ファイル番号の例として，ここまで何度も登場してもらったカセットテープ上のファイルに再び登場してもらいましょう．カセットへファイルを作るステートメントはおなじみの

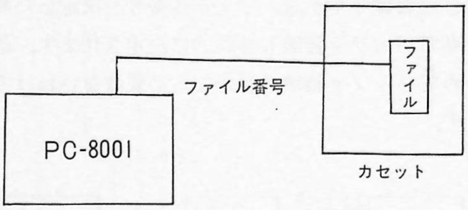
print #-1,〈変数の組〉

で，ファイルを読むステートメントは

input #-1,〈変数の組〉

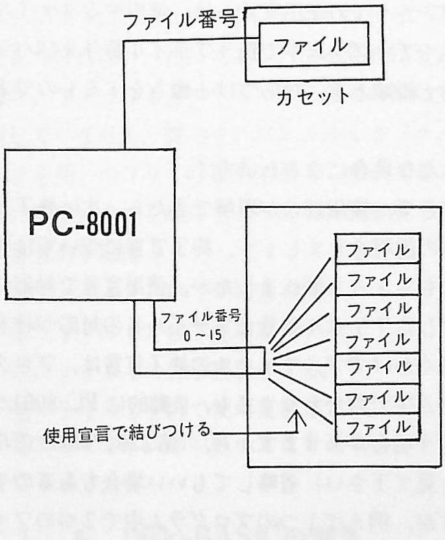
でしたね．この2つのステートメントの#-1の部分に注目して下さい．これがファイル番号にあたるわけです．カセットでファイルを扱う時には，ファイルは1個しか使用できず，そのファイル番号が－1に固定されているわけです．さらに，カセット上のファイルでは，ファイル名は使えませんから，ファイル名とファイル番号の対応づけをする使用宣言と，対応づけを解く終了宣言は必要ないことも理解できるでしょう．

ここまで明らかになったことを，まとめてみます．PC-8001が，ディスクが接続されていない状態で使えるファイルは，カセットテープ上のファイルだけで，ファイル番号は－1です．この状態は下のように書けます．



一方，PC-8031(3032)が接続されると，ディスク上のファイル群が使用できることになります．このファイル番号は可変ですが，最大で1～15までです．また，それぞれのファイル番号に対するファイルも固定されておらず，使用宣言で結びつけて，終了宣言で切りはなすという方法がとられています．

この状態は下のように書けるでしょう．



次に説明するのは使用宣言の動作の2番目で，ディスケットへ書き込んだり，ディスケットから読み出したりするための領域を確保する動作についてです．前に説明したファイル名の宣言は，私たちが実際に行なわなければならない，言わば目に見える動作でしたが，この動作は目に見えない動作で，Disk　BASICがすべて行いますから，興味のない人は読まなくても結構です．

前にディスクへの読み書きは，256バイトでしか行えないことを書きました．一方で，我々人間がデータを扱うのはファイル単位で，更にファイルはレコード単位で読み書きすることも書きました．先に書いた256バイトをデータの物理的単位とすれば，ファイル（レコードでもかまいません）はデータを扱う際の論理的単位と言えるでしょう．

論理的単位と物理的単位が一致していればいいのですが，大抵の場合うまくいきません．ファイルにデータを書き込む時，書き込むデータの長さを計算して256バイトに調整することを好む人はいないと思います．

そこで，Disk-BASICは，自動的に物理的単位を論理的単位へ，論理的単位を物理的単位へ変換してくれるようになっています．データの書き込み，読み出し用の領域というのは変換の手段として用いられる領域でのことで，バッファ（Buffer）領域と呼びます．

バッファ領域を使って，どのように物理的単位と論理的単位の変換を行うか簡単に説明しましょう．詳しい話は第7で参照して下さい．バッファ領域とはディスケットへの物理的な読み書きの単位分の長さ（256バイト）をもった領域のことで、論理的な単位で読み書きされるデータはすべてこのバッファ領域に対して行われるわけです．最終的にバッファ領域が一杯になるか，ディスケットへの書き込み，読み出しの指示が来た時に，バッファ領域の内容をディスケットへ書くとか，次の物理的単位のデータをディスケットから読むとかするようになっています．

とにかく，物理的なデータの単位と論理的なデータの単位のギャップを埋めるために，それぞれのファイルについてバッファ領域が必要な

わけです．

ここまで読めば，もうわかったと思いますが，使用宣言では，使用を宣言されたファイルについてバッファ領域を割当てるのです．〝割当てる〞について注意して下さい．〝準備する〞ではありません．割当てる時には既にバッファ領域は準備されているのです．いつ準備するのでしょうか？Disk BASICは常時使用できるファイルの最大数である15個のバッファ領域を準備しているのでしょうか？違います．そんな無駄なことはしません．

気付いた人もいるでしょう．先に書きましたが，システムをリセットした時に質問されます．

How Many Files (0-15)?

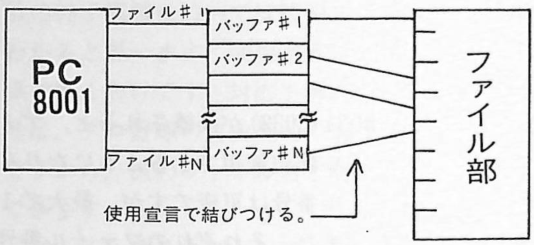
この質問に答えた数だけバッファ領域が準備されるのです．3と答えれば，バッファ領域は3個分確保されるわけです．バッファ領域は，使用するファイル1個につき1個必要ですから，バッファ領域が3個分しかなければ，ファイルは同時に3個までしか使用できません．

実を言えば，「バッファ領域を割当てる」の〝割当てる〞にも少々語弊があります．〝割当てる〞と書くと，使用宣言されたファイルに対して，準備されたバッファ領域のうちどれが選ばれるのか，まるで〝くじ引き〞で決まるような感じがしますがそうではありません．

バッファ領域は，ファイル番号につき1個ずつ定まっているのです．したがって ファイルの使用宣言でファイル番号を指定すると，即座にどのバッファ領域が使用されるか決まるわけです．又，ファイル番号も，準備されていないバッファ領域に対応した番号は使えないことになります．

以上の2点がデータファイルの使用宣言の動作です．2つの動作の説明を理解した人ならすぐわかると思いますが，動作が2つあるといっても，実質的には1つの動作で済むわけです．

なぜなら，バッファ領域はファイル番号に対応した領域ですから，ファイル番号を決定した時点でバッファ領域も自動的に決定されます．改めてバッファ領域を割合てる必要はないわけです．



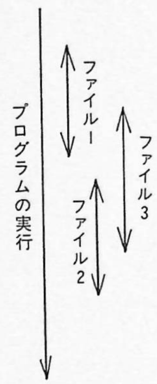
バッファ領域まで考慮してPC-8001のファイル構成を図示すると上のようになるでしょう．

使用宣言の動作を理解したところで，もう一度使用宣言の動作をまとめてみましょう．

ファイルの使用宣言とは，使用するファイルと，プログラム中で扱うファイル番号及びバッファ領域とを，対応づける働きをするものである．

こんな具合になるわけです．

さて，使用宣言が理解できたら，次は終了宣言の説明をしましょう．終了宣言については先にもちょっと触れましたが，使用宣言で対応づけしたファイル番号とファイル名の対応づけを解くわけです．ファイルの終了宣言は，プログラム中で行なわなくても，自動的にPC-8001が行う場合がありますから，（第2章，closeの項を見て下さい）省略してもいい場合もあるのですが，例えば1つのプログラム中で2つのファイルを使用して，その2つのファイルの使用時間が重複しない時，2つのファイルを同一のファイル番号で使用できます．こんな時には，先に使うファイルの終了宣言をして，後で使うファイルを，最初のファイルと同一のファイル番号で使用宣言できるわけです．



具体的に説明しましょう.

上図を見て下さい. 図は3つのファイルを使うプログラムの例で，それが実行された時の使用時間を示しています.

単純に考えると，ファイル1，ファイル2，ファイル3に，それぞれ#1,#2,#3のファイル番号をつければいいわけですが，図を見れば分る通り，ファイル1とファイル2の使用時期は重複していません. 従って，ファイル1とファイル2を同一のファイル番号で扱うことができるわけです. そのためのファイルの使用宣言と終了宣言の手順は

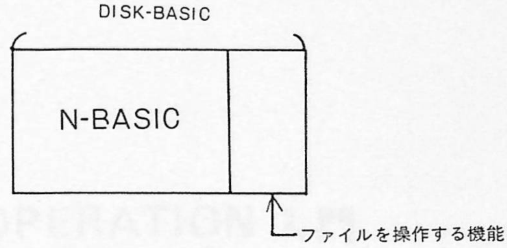
①ファイル1を#1で使用宣言する.
②ファイル3を#2で使用宣言する.
③ファイル1の終了宣言をする.
④ファイル2を#1で使用宣言をする.
⑤ファイル2と3の終了宣言をする.

となります.

1．5　Disk-BASICの働き

これまでファイルの構造について書いてきましたが, Disk-BASICの主な仕事はまさにそのファイルの構造を実現して，操作することなのです.

主な仕事といっても, Disk-BASICがファイルの操作しかできないわけではありません. N-BASICの機能はそのまま引きついでいるのです. Disk-BASICは下図の様に，N-BASICにファイルを操作する機能を付加したBASICのことなのです.



N-BASICについては, PC#1で説明してありますので, このPC#2ではファイルを操作する機能についてだけ説明しているのです.

さて，ファイルの構造を実現し，操作するのがDisk-BASICの働きだと書きましたが,「1-2ファイルの種類」のところで説明しましたように，ファイルにはいくつかの型があります. Disk-BASICもファイルの型によって，その操作法を選択しなければならないことは，容易に察することができるでしょう.

2章のコマンド・ステートメント一覧表を見て下さい. ディスクのファイルに対する操作は，この表中のコマンド，ステートメントを使用して行うわけですが，御覧の通りそれぞれのコマンドなステートメントは，その操作対象のファイルの型が定まっています.

私たちは，表中のステートメントを使ってプログラムを作り，ファイルの操作を行うわけです.

2 OPERATION入門

第1章では，Disk BASICの基礎的なことを説明しましたので，この章では実際のオペレーションのときに出会うコマンドやステートメントひとつひとつについての一応の説明と使い方の簡単な例を示すことにします．さらに詳しい説明や応用は第3章以降で行なうことにしましょう．

2．1　Diskコマンド，ステートメントのあらまし

ディスク関係のコマンドやステートメントは何種もあり，一度に覚えようとしても意味が混乱してしまいますから，その用途別に区分して覚えていくことにしましょう．第1章でファイルをランダムとシーケンシャルの2つに区分しましたが，コマンドやステートメントも，シーケンシャル・ファイルの操作に用いるものと，ランダム・ファイルの操作に用いるものというように区分することができます．このようにして区分したものを表2-1に示します．FILEの

- ファイル
  [MOUNT KILL REMOVE DSKF FILS ATTR$ NAME FORMAT]
  [DSKI$ DSKO$]
  - プログラムファイル
    [SAVE LOAD RUN MERGE]
  - データファイル
    [OPEN CLOSE SET FPOS LOC]
    - シーケンシャルファイル
      [INPUT#　PRINT# PRINT# USING EOF INPUT# LINEINPUT#]
    - ランダムファイル
      [FIELD LSET RSET PUT GET MKI$ MKS$ MKD$ CVI CVS CVD LOF]

表2.1

ところに分類されるものは，プログラムファイル，データファイルの双方のファイルに対して共通に使われるもので，ドライブ単位で作用するものも含めておきました．プログラムファイルに分類したものは，プログラムファイルに対してのみ使われるものです．データファイルは，シーケンシャルファイルとランダムファイルと

に区分されるので関数やステートメントも2つに分けられています．本章を一通り読めば，この表を見るだけで各ステートメントの意味と使い方を思い出すことができるようになるでしょう．

2．2　ファイル一般に対して働くもの

2．2．1　MOUNT

第1章でも説明したように，mount はディスケットの使用開始の宣言をするコマンドです．mount を行なわないと，以下に説明する全てのコマンドやステートメントは(DSKI$とDSKO$を除いて) 使うことができません．mount を行なう手順は次のとうりです．

1．ドライブにディスケットを正しい方向に入れる．
2．MOUNT コマンドを入力する．

このときディスケットは，ディスケット上に記録されているファイルの配置表 (FAT，詳しくは7章を参照) を読み込む動作をするので，必ずディスクユニットのドライブの中にディスケットが正しく装着されていなければなりません．ディスケットが入っていなくても，BASICは入っているつもりで配置表を読もうとします．この時，STOP キーは効きませんから，必ず正しくディスケットを入れたことを確認してから mount コマンドを入力して下さい．たとえば，ドライブ1にディスケットを入れて mount する時には，

mount　1

と入力します．複数のドライブを MOUNT する時には，ドライブ番号をカンマで区切って指定することができます．ドライブ番号を省略したときには，その時 PC につながっている全てのディスク(2台または4台)を mount します．この場合にも，全てのディスクにディスケットを入れたことを確認して下さい．

一度 mount を行なったならば，そのディスケットは，次に説明する remove を行なうまでは，ディスクから取り出してはいけません．また remove を行なわずにもう一度 mount をしようとすると，Disk already mounted のエラーになります．このエラーがでた時は，それ以前に remove をし忘れているということですから，必ず remove をして下さい．

2．2．2　REMOVE

remove は，mount と逆にディスケットの使用の終了を宣言するコマンドです．remove は，mount でメモリに読み込んだ配置表 (FAT) をディスケットに書き込むという大変重要な仕事を行ないますから，ドライブからディスケットを出す時には，必ず remove をしなくてはなりません．remove をしないでとり出すとどうなるかなどの詳しいことは，7章をみて下さい．

remove も mount と同じように，ドライブの番号を指定します．たとえば，ドライブ2に入っているディスケットの使用が終了して取り出したい時には，

remove　2

と入力します．ドライブ番号が省略された時には mount してある全てのディスク中のディスケットを remove します．この時も，BASIC はディスケットが入っているものと思って働きますから，ディスケットが入っているかに注意してください．

一度 remove したのちに，そのドライブに対してディスクコマンドやステートメントを実行すると，Disk not mounted のエラーになります．

2．2．3　FILES, LFILES

FILES コマンドは，ディスケット中にどんなファイルが入っているかを表示します．たとえばドライブ2に入っているディスケットの場合

には，FILES　2と入力します．ドライブの指定が省略された場合にはドライブ1とみなされます．FILESの例を示しましょう．

```
files2
format.     1     backup.     1
copy  .     1     prin25.bas  1
DEMO  .bas  2     fdump .     1
test   asc  1
Ok
```

表示には，ファイルの名前，振張子とそのファイルの大きさが示されます．ファイル名6文字と拡張子3文字との間には，1文字分の間がとられ，非アスキー形式の場合にはピリオドが打たれます．アスキー形式の場合には間はスペースです．上の例ではアスキー形式になっているのは，拡張子ascを持つファイルTESTだけだということがわかります．

ファイル名の右に表示される数字は，そのファイルの大きさを示しますが，この単位はクラスタ数です．クラスタについては，7章で説明しますが，ここでは1クラスタは2Kバイトだということだけを覚えておいて下さい．先ほどの例ではDEMO. basが4Kバイト，その他のファイルが2Kバイトの大きさであるということになります．

LFILESは，FILESと同じ働きをしますが，その出力をプリンタに対して行なう点だけが異なります．

ディスケット中にファイルが存在しない時には，FILESをとっても何も表示はされません．

2．2．4　NAME

NAMEは，ファイルの名前をつけ変えるコマンドです．NAMEは次のように用います．

NAME　〈古いファイル名〉　AS　〈新しいファイル名〉

たとえば，NAME〝ABC″AS〝XYZ″とすると，ABCという名前のファイルはXYZという名前に変わります．

ここでファイル名の指定の仕方について説明しておきましょう．ファイル名の指定はNAMEの他にもloadやsave，openなどのいくつかのコマンドやステートメントでも必要ですが，指定方法はどれもみな同じです．

まず第1にファイル名は，文字型でなければなりません．名前を〝でくくるのはそのためです．文字型であれば文字変数であってもかまいません．たとえば，NAME A$ AS B$とすることもできます．ただし，名前がヌルストリングである場合には，Bad file nameのエラーになります．ファイル名は6文字までで，その後にピリオドをつければ，さらに3文字まで拡張子の指定も行なえます．また，ドライブの指定もDisk　BASICではファイル名の指定に含めて行ないます．たとえば，ドライブ2のABCというファイルを指定するには，

〝2：ABC″

となります．ドライブの指定が省略された場合には，ドライブ1が指定されたものとみなされます．ですから，先ほどの例は，正確にいうならば「ドライブ1のABCという名前のファイルを，XYZという名前に変える」ということになります．もしファイルABCがドライブ2にあるならば，

NAME　〝2：ABC″　AS　〝2：XYZ″

としなければなりません．ドライブの指定は新しいファイル名の方は省略して，

NAME　〝2：ABC″　AS　〝XYZ″

とすることができます．しかし，次のような場合には，Rename across disksのエラーになります．

NAME　〝ABC″　AS　〝2：XYZ″

NAME　〝2：ABC″　AS　〝3：XYZ″

なぜならば，NAMEは異なるドライブにわたっ

て名前の指定がまたがってはいけないからです.

また, 古いファイル名が存在しない時には File not found, 新しいファイル名がすでに存在している場合には File already exists のエラーになります.

NAME はファイルの名前を変えるだけですから, ファイルの内容は全く変わりません. ファイルを本にたとえるならば, NAME はその書名を変更する働きをするわけです.

2. 2. 5 KILL

kill はファイルを削除するコマンドで, 次のように使います.

kill 〈ファイル名〉

KILL では一度にいくつものファイルを削除することはできません. また, 書き込み禁止属性 (SET及び3章を参照) のつけられたファイルは kill することはできず, File write protected のエラーになります. 書き込み禁止属性のついたファイルを削除するには, まず SET で属性を解除してから, kill しなければなりません.

ファイル名の指定のときにはドライブの指定に注意しましょう. ドライブ1と2に同じファイルがあって, 2の方だけを削除したい場合などに指定を忘れると, 1の方が削除されてしまいます.

kill をすると, それまでそのファイルの使用していたディスケット上の領域が解放されて他のファイルを作るのに使えるようになります.

2. 2. 6 DSKF

DSKF は, ディスクの未使用領域の大きさを値とする関数です. ちょうど FRE のディスク版にあたる関数で, 値はクラスタ単位です. DSKF を使って, kill によってファイルの使用していた領域が解放されることを確認してみましょう.

```
files2
format.      1     backup.      1
copy  .      1     prin25.bas   1
DEMO   .bas 2     fdump  .      1
test    asc 1
Ok
print dskf(2)
 54
Ok
kill"2:DEMO.bas"
Ok
print dskf(2)
 56
Ok
```

2. 2. 7 SET, ATTR$

ファイルには属性をつけられることを KILL のところでほのめかしましたが, SET はそのためのものです. 属性については3章でも触れますが, 書き込み禁止とリード・アフタ・ライトの2種類があり, SET を使って次のようにしてつけます.

SET 〈ファイル名〉, 〈属性〉

属性は P (書き込み禁止) と R (リード・アフタ・ライト) で表わして指定します. この指定は必ず大文字で, また文字型になるように" でくくらねばなりません.

SET "ABC", "P"

この例ではファイルABC に書き込み禁止属性がつけられます. save や open によって作られたばかりのファイルには属性は何もついてませんから, 必要ならば SET で属性をつけます. 属性にPとR以外を指定するとそれまでついていた属性は解除されます. 一度ファイルに属性をつけるとその解除か変更を行なうまで変わることはありません.

ATTR$ はファイルにどんな属性がついているかを調べるための関数です. 値は3文字の文字列で与えられ, 属性が無いときには, スペース3つから成る文字列を与えます.

```
set"2:test.asc","P"
Ok
print attr$("2:test.asc")
 P
Ok
```

2．2．8　FORMAT

市販のディスケットを買ってきてPCで使おうと思っても，そのままでは使うことができません．これは，ディスケットのフォーマットがPCの使う型式と異なるためです．(フォーマットについては6章参照)これを，PCの使えるフォーマットに変更するのがFORMAT文です．FORMAT文は次の型式で用います．

FORMAT　〈ドライブ番号〉

FORMAT文を実行すると，指定したドライブに入っているディスケットがフォーマッティングされますが，そのデータは失なわれてしまいますから注意して下さい．市販のディスケットはFORMAT文を実行しただけでは，ディスクBASICで使うことはできません．FORMAT文はその下準備をするだけなのです．実際にDisk BASICで使えるようにするには，システム・ディスケット中のFORMATプログラムをRUNさせる必要があります．このFORMATプログラムは，その中でFORMAT文を実行してくれるので，市販のディスケットを買ってきたならば，FORMATプログラムを実行するだけで充分です．

2．3　プログラムファイルに対して働くもの

2．3．1　SAVE

SAVEは，プログラムをディスクに保存する命令です．ちょうどカセットテープのCSAVEと同じです．SAVEには，アスキー形式と非アスキー形式があることは1章でも説明ました．それぞれの場合のSAVEのしかたは次のようになります．

アスキー形式のとき：SAVE〈ファイル名〉,a
非アスキー形式のとき：SAVE〈ファイル名〉

指定された名前のファイルがすでに存在する場合には，自動的にそのファイルを削除してからセーブを行なうようになっていますから名前の指定には注意してください．

ディスケットにプログラムをセーブするためのあき領域が足りなかった場合にはDisk fullのエラーになります．このエラーが出たならば，不必要なファイルをkillするか，プログラムを短くするかしてsaveし直さなければなりません．それでもだめなら，別のディスケットにsaveすることになります．

2．3．2　LOAD

SAVEによりディスクに作ったプログラムファイルをメモリ中にロード(読み出し)します．

load　〈ファイル名〉

SAVEの時には，アスキー形式か非アスキー形式かによってファイル名の後にaを付けましたが，loadではロードするファイル形式のいかんにかかわらずファイル名の指定だけでよいのです．なぜならばBASICがファイルの形式に応じたロードを自動的に行なってくれるからです．

loadを実行すると，その時メモリ中に入っていたプログラムや変数は全て失なわれて，指定したプログラムファイルが入ります．すなわち，newとプログラムのロードの2つの仕事をするわけです．またloadをすると，その時点でオープンされていたファイルは全てクローズされてしまいます．この事については，CLOSE文のところで詳しくふれます．

loadにはRオプション付きの

load　〈ファイル名〉，R

という型式がありますが，これは

run　〈ファイル名〉，R

と全く同じですので次のRUNのところで説明しましょう．

loadはプログラムファイルをロードするためのものです．指定されたファイルがデータファイルであった場合にはその動作は保証されないので注意して下さい．

2．3．3　RUN

Disk　BASIC では，RUN でファイル名の指定を行なうことができます．

run　〈ファイル名〉

この動作は，まずファイルをロードして，最初の行から実行を始めるのと同じですので

load　〈ファイル名〉

run

と等しいことになります．実行を開始する行の指定は行なえません．また load と同じように，それまでメモリ中にあったプログラムや変数は失なわれてしまいますし，オープンされていたファイルは全てクローズされます．但し次に述べるRオプション付きの場合には，ファイルのクローズは行なわれません．

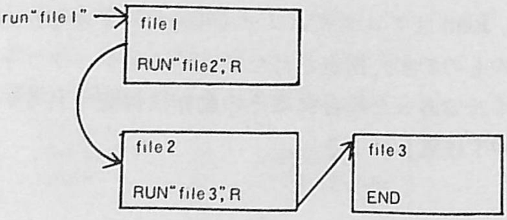
Rオプション付きの RUN とは次の型式です．

run　〈ファイル名〉，R

動作はほとんど同じで，ファイルがクローズされない点だけが異なります．

このRオプションはプログラムのチェインを行なうのに用いられます．具体例は 4 章で示しますが，手短かには次のようなことになります．

ディスクの特長は記憶容量が大きい事でしたから，メモリにはいりきれないような大きなプログラムでも，実行に必要な部分だけを LOAD してプログラム全体の実行していくことができれば，プログラム全体を処理が可能となります．そのためには，プログラムをいくつかのモジュールに分けて，各々をプログラムファイルとしておきます．各モジュールの最後には load または run をRオプション付きで用いることにより，次のモジュールに制御を移すようにするのです．



このように，チェインはファイルへの GOTO に相当するものです．ファイルがロードされる前のプログラムは NEW されてしまいますから各モジュールの行番号が衝突してもかまいません．しかし，ファイルへの GOSUB のようなことはできないことになります．また NEW により変数の値が失なわれてしまいますから，モジュール間でのデータの受け渡しを行なうには工夫が必要となります．この方法としては次の 3 つが考えられるでしょう．

1．BASIC で使用するメモリの上限を CLEAR 文で制限しておき，それ以降のメモリ空間に PEEK，POKE することにより受け渡しを行なう．
2．データを受け渡すためのファイルを準備しておき，そのファイルを介して受け渡しを行なう．
3．ファイルの使用するバッファを利用してデータの受け渡しを行なう．

1 と 3 の方法は，NEW されることにより，値の失なわれることのないメモリ空間を使おうという方法です．それに対し 2 の方法は，値の失なわれることのないファイルを使おうというものです．ファイルとしてシーケンシャルファイルを使う場合には、ファイルを先頭から読み直さねばならないために，引き渡す側のクローズと受け取る側の OPEN が必要となりますが，ランダムファイルを使う場合には，ファイルがオープンされたままになりますからその必要がなくなります．たとえば変数AとBの値を PROG 1 からPROG 2 に渡すには次のようになるでしょう．

i）シーケンシャル　ファイルの場合

```
10  REM  PROG 1
20  OPEN  "PERM" FOR OUTPUT AS  #
1
25  INPUT  A, B
```

```
30  PRINT  #1, A, B
40  CLOSE  #1
50  RUN  "PROG 2", R

10  REM  PROG 2
20  OPEN  "PERM"  FOR  INPUT AS
 #1
30  INPUT  #1, A, B
40  PRINT  A, B
45  CLOSE  #1
50  END
```

ii）ランダム　ファイルの場合

```
10  REM  PROG 1
20  FIELD  #1, 4  AS A$, 4 AS B$
30  OPEN  "PERM"  AS #1
40  INPUT  A, B
50  LSET  A$=MKS$ (A)
60  LSET  B$=MKS$ (B)
70  PUT  #1, 1
80  RUN  "PROG 2", R

10  REM  PROG 2
20  FIELD  #1, 4 AS A$, 4 AS B$
30  GET  #1, 1
40  PRINT  CVS (A$), CVS (B$)
50  CLOSE  #1
60  END
```

この例では，まだ説明していないステートメントがいくつもでてきましたがこれらについては2．4節で触れます．受け渡しの方法の1と3については，3章以降で説明されるでしょう．

2．3．4　MERGE

プログラムのチェインでは，1つのプログラムをいくつかのモジュールに分解することを考えましたが，MERGEはこれと逆にいくつかのモジュールをまとめて1つのプログラムを作るのに使います．頻繁に使用するようなサブルーチン等は，ライブラリとしてプログラムファイルを作っておき，必要となったときに，MERGEによりメインプログラムと結合を行なうのです．本書ではプログラムの実行結果をプリンタに出力していますが，どのようにして出力させているかというとスクリーンの内容をプリンタに出力するプログラムを1つ作っておき，実行例を示すときには，このプログラムとメインプログラムとをMERGEして実行させているのです．

MERGEは，このようにして，メモリ中にあるプログラムと，プログラムファイルとを一諸にして1つのプログラムとします．使い方は次のとおりです．

MERGE　〈ファイル名〉

MERGEするファイルはアスキー型式でなければなりません．非アスキー型式のときにはBad file modeのエラーになります．またチェインの時と異なり，メモリ中のプログラムと指定するファイル中のプログラムには同一の行番号がないようにしなければなりません．同一の行番号があった時には，ファイルの方のものが入ります．つまり，MERGEの動作は，指定したファイル中のプログラムをPCがタイプしてくれると考えれば良いでしょう．

MERGEではメモリ中のプログラムとファイル中のプログラムとの2つの結合しか行なえませんから，3つ以上のモジュールの結合を行なうには，MERGEを複数回用います．行番号の衝突の問題を避けるには，ライブラリとしてファイルにしておくプログラムは，大きな行番号（たとえば1000）から始まるようにしておき，MERGEしたならばRENUMするようにしれば良いでしょう．MERGEした結果はメモリ上

にできることにも注意して下さい．何度も続けて大きなファイルをMERGEしているとメモリが足りなくなります．プログラム間の変数名の衝突にも注意して下さい．

2.4 データ ファイルに対して働くもの

2．4．1 OPEN

データファイルを使うには，そのファイルを参照する前にその使用の開始宣言が必要です．ファイルを本にたとえて考えてみると，ディスクは，本がたくさん入っている図書館のようなものです．あなたは，座席にすわっていて，見たい本があればBASICにその本を持って来てくれるように頼むわけです．OPEN文はこのようにディスケットからあなたの注文したファイルを探し出してくれる働きをします．OPENをして初めて、あなたはそのファイルを参照することができるようになるのです．

この図書館は少々変わっており，本を読むだではなく，本に書き込みをすることも許してくれます．また，新しく本を作ることもできます．詳しい注文のしかた（OPEN文の使い方）については2・5節と2・6節で説明しましょう．

ここでは，この図書館の貸し出し規則について述べておきます．注文によってBASICはあなたの机まで本を持ってきてくれます．また，本を作るときには，表紙だけで内容は白紙の本を持ってきてくれると思ってください．机には広さの制限があるので，一度に借り出せる本の冊数は制限されます．この制限は，あなたが，机の大きさをあらかじめ予約したときに決まります．すなわち，大きな机を予約しておけば，それだけ多くの本を広げることができますが，小さな机では，そんなに多くの本は無理です．また，準備されている机で最も大きいものは，15冊までの本を広げられるようになっています．この予約は，Disk　BASICをスタートさせた時に，質問してくる，"How many files（0-15）？"にどう答えるかによってなされます．この時に15と答えていれば15冊まで借りられますが，0と答えた人には1冊も貸してくれません．

2．4．2 CLOSE

OPENは本の借り出しに相当するものでしたが，CLOSEは返却に相当するものです．実際の返却手続き（CLOSE文の使い方）については後で述べますが，借りた本は必ず返さなければなりません．いつまでも返却を忘れていると強制的に取り返されてしまいます（自動的にCLOSEされる）．Disk　BASICがこのような手段にでるのは次の場合です．

1．END文，NEWを実行したとき．
2．プログラムの文面に変更を加えたとき．
3．REMOVEを実行したとき
4．RUN，LOADを実行したとき．

但し，すでに説明したように，Rオプション付きのrun，loadの時にはCLOSEは行なわれません．

CLOSEによって返却をおこなうと，その分だけ机の上が広くなるので，またOPENによって別の本を借りてくることができるようになります．ですから，最低一冊分のスペースがあれば借り出しと返却（OPENとCLOSE）をくり返すことにより，何冊もの本（いくつものファイル）に対して読み書きが行なえることになります．しかし，一度に何冊もの本を広げたり，一冊の本の内容を別の本に書き移したりするためには、一冊分のスペースでは足りないことになります．

本の返却手続き（CLOSE文の使い方）としては，本の置いてある場所を指定します．つまり，「そこに置いてある本返してきてよ．」とBASICに頼むわけです．OPENのところでは説明しま

せんでしたが，借り出すときにも，〝こういう題名の本をもってきて机のこの位置においてくれ.〟というように場所を指定しなければなりません．OPEN してから CLOSE するまでの間，その本の置いてある場所を変えることは，できません．また，本を読んだり書いたりする時には，この位置を指定することになります．場所の指定は番号で行ないます．すなわち，机の上に線が引いてあって本を置く場所が区切られており，各区画には番号がふってあると思えばよいでしょう．

結局 CLOSE 文の使い方は次のようになります．

「1番のところにある本を返して下さい.」

CLOSE ＃1　または，CLOSE 1

1度に何冊もの本を返すこともできます．

「1番と2番のところにある本を返して下さい.」

CLOSE #1, #2　または　CLOSE　1，2

#記号はあっても，無くてもかまいません．番号の指定をしないと机の上にあった全ての本（ファイル）を片づけられてしまいます．

この番号のことを，ファイル番号と呼びます．

2．4．3　SET

SET は2．2節でもでてきましたが，ここで再びの登場です．2度もでてくるのはその使われ方が異なるためです．2．2節の SET はファイルに属性をつけるもので，いわば本そのものに対して書き込み禁止のラベルをつけるものでした．ここで説明するのは，SET のもう1つの使い方で，ファイル番号すなわち机上の場所を指定します．

SET　〈ファイル番号〉，〈属性〉

たとえば　SET　#1，〝P〟は，「1番の場所にある本に対しては書き込みを禁止するようにする」という働きになります．注意しなければならないのは，本（ファイル）に属性がつくのではなく，場所（ファイル番号）に属性がつくのだということです．今1番にP属性がついていると，そこの本を CLOSE により返却しても，その本に書き込み禁止のラベルが貼られるわけではありません．この SET では指定したファイル番号のファイルはオープンされていなければなりません．また，SET された属性は CLOSE すると解除されてしまいます．

2．4．4　FPOS

FPOS はファイルがディスケット上のどの場所にあるかを知らせてくれる関数です．この関数は，普通使わないものなので，ここでは使い方と注意だけを示します．詳しくは7章でもう一度触れます．

FPOS は次のようにして用います．

FPOS　〈ファイル番号〉

返される値は，トラック0のセクタ1を0とする通しの番号です．いわば本が本棚のどこにしまってあるかを教えてくれるような関数です．FPOS の返す値をFで表わせば，トラック番号Tとセクタの番号Sは次のようにして求められます．

T=F　¥　16 : S=F mod 16+1

注意しなければならないのは，FPOS の返す値は，ファイルがオープンされている間だけ有効だということです．また，OPEN した直後の FPOS の値は，そのファイルの入っていた最初の位置を示しますが，1度でもディスクへのアクセスが行なわれると値は変化します．この値は，次にアクセスするべきディスク上の位置を示すものなのです．詳しくは7章をみて下さい．

また，FPOS は指定したファイルがどのディスクに入っているかは教えてくれないことにも注意してください．

2．4．5　LOC

LOC はFPOS の親類の関数で，次に読み書きするレコードの値を返すものです．LOC の示すレコードのある場所を与えるのがFPOS にあたるのです．LOC の詳しい説明をするには，ここまでは知識不足なので２．６節及び３章に譲ることとします．

2．5　シーケンシャル　ファイルに対して働くもの

ここでは，シーケンシャルファイルの操作に用いるステートメントや関数について説明します．シーケンシャルファイルについてのおさらいをしておいてください．

2．5．1　OPEN

シーケンシャルファイルは本にたとえていうならば巻物に相当します．巻物を持って来てもらう（OPENする）場合には３種類の注文の仕方があります．

1．OPEN　〈ファイル名〉FOR INPUT AS 〈ファイル番号〉

巻物〈ファイル〉を読む場合の注文の仕方を決める命令です．これを行なうと BASIC は，指定した題名（ファイル名）の巻物を探し，指定した場所（ファイル番号）へ持ってきてくれます．指定したファイルが存在しない場合には File not found のエラーになります．

2．OPEN　〈ファイル名〉FOR　OUTPUT AS　〈ファイル番号〉

新しく巻物を作る場合の注文の仕方を決めます．BASIC は指定した題名と同じ題名の巻物があった場合には，それを廃棄 (KILL) してから中身が白紙の巻物を用意してくれます．同じ題名の場合には，白紙の巻物を用意してくれるだけです．用意された巻物は指定した場所（ファイル番号）に置かれます．

3．OPEN　〈ファイル名〉　FOR　APPEND AS　〈ファイル番号〉

ファイルに対して書き足しをする時の注文の仕方です．BASIC はファイルを探してきてその終りの部分を広げて，あなたがすぐに書き足しをできるように準備してくれます．指定したファイルが存在しない場合には File not found のエラーになります．

このように INPUT，OUTPUT，APPEND の３つの注文の仕方がありますが，これを OPEN のモードの指定と呼びます．モードが省略された時には，ファイルが存在すれば INPUT の場合と同じ動作，存在しない場合には OUTPUT の場合と同じ動作になります．

モードの指定は，そのファイルに対して書き込みを行なうか，読み出しを行なうかの宣言ですから，以後はそれに従がわねばなりません．OUTPUT 及び APPEND の場合には着目している部分がファイルの終りにあたるわけですから，読もうとしても，そこには何も書かれていないために，Input past end のエラーになります．INPUT のモードに指定した場合には，そのファイルに対する書き込みを行なっても，BASIC は何のエラーも出しません．しかし，これはエチケット違反であり，その動作は保証されません．ファイルに少々のデータを書いてから CLOSE しても実際には書き込みをしなかったり，ある程度以上を書くと実際に書き込みを行なってもとの内容を破壊することになります．INPUT のモードで OPEN した場合にはこのような危険性が伴ないますから，OPEN したならば，SET でそのファイル番号にはP属性をつ

けることを心がけると良いでしょう.

一度 OPEN したならば, CLOSE するまでは,同じファイル番号を使って,ファイルのOPENを行なおうとすると File already open のエラーになります. しかし, 同一のファイルを異なる場所 (ファイル番号) でOPENすることはできます. (これは INPUT のモードの時だけです)

2.5.2 PRINT#

シーケンシャルファイルの書き込みを行なうには PRINT#を次の型式で用います.

PRINT # <ファイル番号>, <式の並び>

例えば変数Aの値を1番のファイルに書き込むには PRINT #1, Aとします. ファイル番号が-1の場合はカセットへの出力にあたります. このように書式はいたって簡単なのですが, ディスクを効率良く使ったり, 文字列を書き込む場合にはいろいろと細かい注意が必要となります. それらについては, 具体例を示して3章でくわしく説明します. ここでは, PRINT#は, PRINT 文で画面に表示するのと全く同じようにファイルに対して書き込むのだということを覚えておいて下さい.

PRINT#を使うには, 指定されたファイルはオープンされていなければなりません. また, INPUT のモードで OPEN したファイルに対して PRINT#で書き込んだ場合には実際に書き込みの行なわれない場合もあるので注意が必要です. (INPUT のモードで OPENしたファイルに対して書き込みを行なってはならないと考えて下さい)

2.5.3 INPUT#

シーケンシャルファイルからの読み込みを行なうのが INPUT#です. つまり, PRINT# で書かれたデータを読むことになります.

INPUT# は次の型式で用います.

INPUT# <ファイル番号>, <変数の並び>

ファイル番号が-1のときには, カセットからの入力になります. 簡単な例で INPUT# の動作について説明をしておきましょう.

```
list
10 OPEN "test" FOR OUTPUT AS #1
20 PRINT #1,1,2,3
30 CLOSE #1
40 OPEN "test" FOR INPUT AS #1
50 INPUT #1,A,B,C
60 CLOSE #1
70 PRINT A,B,C
80 END
Ok
run
 1             2
 3
Ok
```

INPUT#を使うには, それ以前に読み込みを行なうファイルがINPUT のモードでオープンされていることが必要です. 例では, 10行から30行で作られたファイル test をオープンしています. ファイル test には、数字1, 2, 3がこの順番で書かれていますが, 40行以下でそのことを確かめてみたというわけです. 40行では CLOSE したばかりのファイルを OPEN しているのですが, こうしないとファイルのデータは読めません. このことについて, 説明をしておきましょう.

OPEN, CLOSE の説明ではシーケンシャルファイルを巻物にたとえて説明しましたが, PRINT# と INPUT#の働きを説明するため, カメラのフィルムに置き換えて説明することにしましょう. どちらも〝巻物〟というイメージがシーケンシャルなファイルということを意味します.

10行の OUTPUT モードの OPENは, これから書き込みを行なうということですから, 新しい何も撮っていないフィルムをセットすることにたとえることができます. 20行での PRINT# は, このフィフムに1, 2, 3の値を写す (書き込む) ことになるのですが, フィルム1枚に

は1つの値しか写せないと思って下さい．また，シャッターを押して値を写すと，最近のモータードライブのカメラのようにフィルムは自動的に巻き上げられるとしましょう．したがって20行の PRINT# の動作は次のようになります．

1．シャッターを押して〝1〞を写す．
2．シャッターを押して〝2〞を写す．
3．シャッターを押して〝3〞を写す．

このようにフィルムには1，2，3の値がこの順番で写されたことになります．さて，こうして写された値を読むためには，フィルムを巻き戻す必要があります．シャッターを押すと同時にフィルムは巻き取られていますから，1番最後に撮った値すら読めないのです．40行のOPENは，まさにこのフィルムの巻戻しに相当します．INPUT のモードということは，フィルムの最初のコマが写し出されることだと考えれば良いでしょう．50行のINPUT#文はこうして写し出された値を読み取る働きをすることになります．まず1を読み取り A にしまいます．読み取ると同時にフィルムは巻き上げられると思って下さい．A の次には B にも読み込んで欲しいという要請があるので今出している値2を読み取り，Bにしまいます．同様にして3が読みとられCにしまわれます．このように，シーケンシャルファイルは書かれた順番でしか，読むことのできないものです．さて，ここで3の値を読み取った後のフィルムの状態について注意して下さい．3を読み取った後，フィルムは巻き取られていますが，そこには何も写っていないはずです．なぜならば，20行のプリント文では1，2,3の3つしか写していないからです．この何も写されていないコマを読もうとするとエラーになります．50行を次のように変更して確めてみましょう．

```
50 INPUT #1,A,B,C,D
run
Input past end in 50
Ok
```

Dの値を読もうとする時に何もないのでエラーとなったのです．これはちょうど，READ文で DATA 文のデータを読むときに Out of data のエラーがでる場合と同じです．一般にファイルのデータを読む場合には，そのファイルの大きさが未知であることが多いので，ファイルの終りに達したかどうかを知る手段が必要です．そのための関数が次に説明する EOF です．

2．5．4 EOF

EOF はファイルの終り (End of File) に達したかどうかを調べるための関数です．終りに達していたならば真 (-1)，達していなかったならば偽 (0) の値を返します．EOF の返す値が真になったということは，その次にファイルを読もうとすると Input past end のエラーになるということを意味します．EOF の使い方の簡単な例を示しておきましょう．

```
list
10 OPEN "data" FOR INPUT AS #1
20 SUM=0
30 INPUT #1,A
40 SUM=SUM+A
50 IF NOT EOF(1) THEN GOTO 30
60 CLOSE #1
70 PRINT SUM
80 END
Ok
```

この例は，〝data〞という名前のファイルに書かれている値を読み，その合計を求めるものです．50行でファイル Data が終わりになったかをチェックし，終りでなければ30行に戻って次のデータを読み込むわけです．EOF 関数はこのような型の入力のループで用いますが，このループを使うときは，ファイルに少なくとも1つ

のデータが書かれていなければならないことに注意して下さい．データが無いかも知れない時には，EOF の条件判断を INPUT #の前に置かなければなりません．

EOF は，このようにシーケンシャルファイルを読む場合に必ず登場するものです．ファイルの終りの印は，ファイルに対して書き込みを行なって CLOSE した時につけられるものと考えて下さい．

2. 5. 5 LINEINPUT#

LINEINPUT# は，INPUT# で文字列を読む場合に，カンマや引用符が読めないという不便さを補うためのものです．LINEINPUT# では CR／LF で区切られるか，ストリングの長さの制限である255文字に達するまでの全ての文字列の読み込みを行ないます．具体的な使い方については 3 章で示します．普通の LINEINPUT では，カーソルコントロール等のコントロールコードは読み込みませんが，LINEINPUT# では読み込まれることに注意して下さい．

2. 5. 6 INPUT$#

INPUT$ はキーボードから指定した長さだけの文字を読み込むものでしたが，ファイル番号の指定をつけると，指定したファイルから指定した長さだけの文字を読み込む働きをします．

INPUT$（〈長さ〉,#〈ファイル番号〉）たとえば INPUT$（1,#1）は 1 番のファイルから 1 文字だけ読み込みます．この 1 文字だけ読むという処理はテキスト処理をする時には頻繁に用いられるものです．2. 5. 8で示すプログラムで実際に使われていますので参照して下さい．

2. 5. 7 PRINT#USING

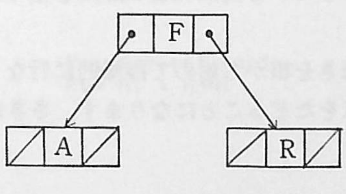
PRINT　USING のディスク版です．動作も全く同じで，USING の前にファイル番号の指定をつけるだけで，ファイルに対して書式の指定どうりの出力を行ないます．実際のデータ処理では，処理した結果を全てファイルに出力しておいて，後でまとめてプリンタに出力するということがしばしば行なわれますが，そのような場合に，PRINT#USING が用いられます．

2. 5. 8 シーケンシャルファイルを用いたプログラム例

ここで紹介するプログラムは，入力ファイル中の単語を調べ，どの語が何回表われたかをアルファベット順に並べた表にして出力するもので，文字を扱うシーケンシャルファイルの良い例になっていると思います．数値データのシーケンシャルファイルについては 3 章を見て下さい．さて出力をアルファベット順にするためには，全ての語を配列の中に入れておき，最後にそれを整列（ソーティング）するのが 1 つの方法ですが，ここで示す方法は 2 分検索という方法で，木が常にアルファベット順に並んだ語を表わすように入力した語を木に加えていく方法です．まず 2 分木という構造について説明しましょう．

2 分木とは 1 つの項目に他の 2 つの項目をさすポインタを設けたものの集まりのことです．その基本要素は節点と呼ばれ，特に木構造の始点となる節点を根と呼びます．また各節点からみて左の部分を左 2 分木，右の部分を右 2 分木と呼びます．

この 2 分木を使って順序を表現することは，左 2 分木にその節点より順序の小さなものをぶらさげて、右 2 分木には順序の大きいものをぶらさげることにより達成されます．(もちろんこの反対でも良いのですが)．たとえば F を根とするような木で A, F, Rの 3 つの語の順序関係を示せば下図のようになります．

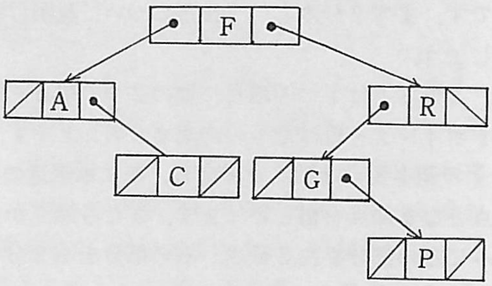


各節点は，左2分木をさすポインタと右2分木をさすポインタを持つのでこれを LEFT, RIGHT と呼ぶこととしましょう．F の LEFT は A をさし，RIGHT は R をさしています．A と R のポインタは何もさすものがないので，これを斜線で表わしておきます．

さて，この構造を使ってある語を探したり，追加したりすることは次の手順に従えば行なえます．

- まず根に着目する．希望する語がこの節点の語よりも大きければ，木を右へ進む．もし小さければ左へ進む．希望する語が見つかれば，めでたし，めでたし．
- 不幸にして，見つからないまま終点に達してしまったならば（ポインタが何もさしていない場合），その語を順序の大小に応じてその節点の右か左に付け加える．

たとえば先ほどの本に C，G，P をこの順番で追加すれば下図のようになります．



このようにして木が完成したならば，結果を出力するために，木をたどらねばなりません．左2分木の方が順序が小さいのでしたから，基本的な考え方は次のようになります．

1．着目している節点の左2分木を全てたどる．
2．着目している節点の語を出力する．
3．着目している節点の右2分木を全てたどる．

この手続きを根から始めて再帰的に行なうと全ての節点をたどることになります．さきほどの木をたどってみると…．

- まず根である F に対して上記の1の仕事を始める．F の LEFT をたどり A につく．
- A に対して1の仕事を行なう．LEFT は何もさしていないので1の仕事は終了．2の仕事を行ない A を出力．続いて3の仕事を始める．RIGHT をたどり C につく．
- C の LEFT は何もさしていないので1の仕事は終了．2の仕事でCを出力．RIGHT も何もさしていないので3の仕事も終了．A に戻る．
- Cの処理が終了したということは，A の3の仕事も終了したということなので F に戻る．
- A から戻ってきたということは，F の1の仕事が終了したということであるから，2の仕事を行なって F を出力する．続いて3の仕事を始める．RIGHT をたどり R に着く．
- R のLEFT をたどり G に着く．R を出力し，RIGHT をたどりP に着く．
- P の LEFT，RIGHT は何もさしていないので P を出力して G に戻る．
- G の処理が終わったので R に戻る．
- R を出力．RIGHT は何もさしていないので F に戻る．
- F の右2分木を全てたどったので F に対する処理は終了．F は根であったから，木を全てたどった!!

本書は，データ構造やアルゴリズムに関する説明を目的としているわけではないので2分木に関する説明はこの程度にしておきます．木構造については Knuth の "Fundamental Algorithms" （サイエンス社「基本算法」）に詳しく書かれていますので参照してください．またアスキーシステムバンク PET#3には木構造を扱った BASIC プログラムが載っているので参考になると思います．

```
100 '*
110 '*    Demo Program for Section 2.5
120 '*
130     CLEAR 3000                                     'Reserve string space
140     SIZE=300                                       'up to 300 words
150     DIM TABLE$(SIZE),LEFT%(SIZE)
160     DIM RIGHT%(SIZE),COUNT%(SIZE)
170     DIM STACK%(50)
180 '
190 '*
200 '*    Main Routine
210 '*
220 '
230     INPUT "file name:";NM$                         'Input file name
240     OPEN NM$ FOR INPUT AS #1
250     AL$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"               'AL$ is alphabet
260     AN$="0123456789"+AL$                           'AN$ is alphanumeric
270     NF=1:R1=0
280 '
290     FIND=0                                         'Clear find flag
300       GOSUB 380                                    'Get word
310       IF FIND THEN GOSUB 510
320     IF NOT EOF(1) THEN GOTO 290                    'Repeat until eof
330     GOSUB 610                                      'Print
340 '
350     CLOSE #1
360     END
370 '*
380 '*    Subroutine: GETWORD
390 '*
400     CH$=INPUT$(1,#1)
410     IF CH$=" " THEN GOTO 400                       'Skip blank
420 '
430     WO$=""
440     IF INSTR(AL$,CH$)=0 THEN GOTO 490              'If CH$ is not in AL$ then ret.
450       WO$=WO$+CH$
460         CH$=INPUT$(1,#1)
470       IF INSTR(AN$,CH$) THEN GOTO 450              'Repeat until CH$ is not in AN$
480       FIND=-1                                      'Set find flag
490     RETURN
500 '*
510 '*    Subroutine: SEARCH
520 '*
530     R=R1
540     IF R=0 THEN TABLE$(NF)=WO$:LEFT%(NF)=0:
          RIGHT%(NF)=0:COUNT%(NF)=1:NF=NF+1:
          GOTO 580                                     'New word ... append
550      IF TABLE$(R)>WO$ THEN R1=LEFT%(R):
           IF R1=0 THEN LEFT%(R)=NF:GOTO 530
                    ELSE GOTO 530                      'Search left part
560       IF TABLE$(R)<WO$ THEN R1=RIGHT%(R):
            IF R1=0 THEN RIGHT%(R)=NF:GOTO 530
                    ELSE GOTO 530                      'Search right part
570       COUNT%(R)=COUNT%(R)+1                        'Find ... increment count
580     R1=1
590     RETURN
600 '*
610 '*    Subroutine: PRINT
620 '*
630     SP=0:R=1
640     STACK%(SP)=R1:SP=SP+1                          'Push R1
650     IF R=0 THEN GOTO 700
660       R1=R:R=LEFT%(R):GOSUB 640                    'Print left part
670       PRINT TABLE$(R);TAB(15);
680       PRINT USING"####";COUNT%(R)
690       R1=R:R=RIGHT%(R):GOSUB 640                   'Print right part
700     SP=SP-1:R=STACK%(SP)                           'Pop R1 : R=R1
710     RETURN
```

まずプログラムリストについて少し説明しておきましょう．リストはプリンタでとったものなので画面にでるものとは字体が異なりますので注意して下さい．特に，O と 0 は慣れないと間違え易いかもしれません．またリストは，プログラムの意味と構造がわかりやすいことを重点としているので注釈がかなり入っていますし，行の頭に空白を入れ段づけも行なっています．（′は REM 文の省略型です）あなたがプログラムを入れるときにそのまま入れてくださっても結構ですが，REM 文や余分な空白は実行には関係のないものですから取り去ってもかまいません．変数名も，プログラムの意味がわかり易いように長いものを使っていますが，BASIC は最初の 2 字しか判別しませんから短くすることができます．

プログラムの先頭では必要な宣言を行なっています．文字型の配列を使うので CLEAR 文で文字領域をとっておきます．ここでは，単語は300種までということに限定して考えることにしましたので3000あれば充分でしょう．配列，TABLE は，単語を入れておくための配列です．LEFT%，RIGHT%は左右のポインタのための配列です．整数型にしておくのはメモリ使用量が少ないからです．この 2 つの配列は，次の節点を示す添字の値を入れることによりポインタとして用います．COUNT%は単語の出現回数を入れておくためのものです．

230行から360行はメインルーチンです．まず230行でファイル名を入力し，240行でそのファイルをオープンします．このようにファイル名の指定には文字変数がよく用いられます．

290行から320行で構成されるループでは，単語をひろってきては 2 分木に入れる作業を繰り返します．EOF まで達するとこのループから抜け出し結果をプリントして終わりです．

サブルーチン GETWORD は入力ファイルから単語を 1 つひろってくる手続きです．ここで単語とは英字で始まり，英字または数字が続く文字列であると決めて処理しています．数字を含むような単語があるのかと思いになるかもしれませんが，BASIC の変数名がそうです．実行例でも示しますが，入力ファイルとしてプログラムファイルを扱うことを前提としたのでこのような規則になっています．

まず400行と410行とで空白の読みとばしを行ないます．ファイル中から 1 文字読み出すのには INPUT$ が使われています．

空白でない文字が見つかったならば，単語である可能性があるので WO$ をヌルにしておきます．ひろった単語は WO$ に入れられるからです．440行では文字が A から Z の英字であるかチェックを行ないます．特殊記号は単語の先頭文字となる資格がないので，ここではねられてメインルーチンに戻ります．この時 FIND の値が 0 （偽）で変わらない事に注意してください．FIND の値が 0 の時には木への追加は行なわれません（310行）．

めでたく単語の先頭文字と認められたならば続いて後続文字がひろわれて WO$ にまとめられます（450行～470行）．英字でも数字でもないのがでてきたら語の区切りとみなされるわけです．こうして無事に単語をひろうことができた場合には FIND が -1 （真）にセットされます．

サブルーチン SEARCH は単語 WO$ が 2 分木の中にあるかを調べる手続きです．見つかった場合にはその出現回数を 1 つ増やし，見つからなければその語を木に追加します．この手続き中では R はポインタの値を示します．ポインタの値はさし示すものがあればその配列の添字の値，何もさすものがない場合は 0 とします．

540行はポインタが何もさしていない場合，つまり，木に追加する場合の処理です．変数 NF にはまだ使っていない添字の値（Next Free）

を保持していますから，TABLE$(NF)に単語の登録をします．新しい節点はまだどこもさしませんから対応する LEFT，RIGHT は0にしておき，出現回数を1回にします．NF を1つ増やせば処理は終了です．

550行は，WO$ の順序が着目している節点の単語よりも小さい場合の処理です．配列 LEFT%の値をひき出し R に代入することにより木を左へたどることを実現しています．もし，LEFT が何もさしていない場合には新しい節点を左にぶらさげる必要がありますから LEFT% (R) に NF を代入しておきます．GOTO で350行に戻ることにより探索は続行されます．

560行では同様に WO$ の順序が大きかった場合の処理が行なわれます．

570行は単語 WO$ が見つかった場合の処理にあたります．580行で R1を1にしているのは，このサブルーチンが2回以上呼ばれた時に正しく根から探索を始めるための操作です．

サブルーチン PRINT は2分木をたどって各節点の単語とその出現回数を出力する手続きです．手続きが再帰的に定義されていたのでプログラムでも再帰呼び出しを用いています．660行で左2分木をたどり690行で右2分木をたどります．手続きから戻るときには木をどのようにたどってきたかに応じてどの節点に戻れば良いかを覚えておく必要があるのでポインタ R の値をスタックにプッシュ，ポップしていることに注意して下さい．

以下に実行例を示しますが，この結果は入力ファイルとしてこのプログラムのプログラムファイルを指定したものです．プログラムがアスキーセーブされていればこのようにしてプログラム中のキーワードや変数名の出現回数を調べることができるのです．このプログラムは小文字は区切り記号とみなすことになっているので注釈中の文字は大文字しかひろわれていません．もし，完全にプログラムのキーワードと変数名だけを処理したければ注釈を LINEINPUT# を使って読みとばす必要があるでしょう．また$, %, #を変数名の最後の文字として認めた方が良いでしょう．

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ     1
AL                     5
AN                     4
AS                     1
C                      1
CH                     8
CLEAR                  1
CLOSE                  1
COUNT                  5
D                      1
DIM                    3
ELSE                   2
END                    1
EOF                    1
F                      1
FIND                   3
FOR                    1
G                      1
GETWORD                1
GOSUB                  5
GOTO                  10
I                      2
IF                    11
INPUT                  4
INSTR                  2
LEFT                   5
M                      1
N                      1
NF                     9
NM                     2
NOT                    1
OPEN                   1
P                      6
PRINT                  3
R                     26
R1                    13
RETURN                 3
RIGHT                  5
S                      8
SEARCH                 1
SIZE                   5
SP                     7
STACK                  3
TAB                    1
TABLE                  5
THEN                  11
USING                  1
WO                     6
```

プログラムに少し変更を加えて小文字や特別な記号を受付けるようにすると一般の文書も処理できるようになります．次に示すのはある本の序文（PREFACE）を処理した結果です．M のところにでてくる人物（？）の名前から本の内容が推察できると思いますが….

```
A                  8
ABOUT              4
ACHIEVED           1
ACROSS             1
ADVENTURES         1
AGE                2
ALBEIT             1
ALL                3
AM                 3
AND               12
ANOTHER            2
ANYTHING           1
ART                1
AS                 2
AT                 1
ATTAIN             1
AUGUST             2
BE                 2
BEAR               1
BECOME             1
BEEN               1
BEFORE             1
BEING              1
BELIEVE            1
BOOK               1
BORING             1
BRING              1
BUT                2
BY                 1
CHILD              1
COME               1
CONTRIBUTED        1
DADDIES'           1
DADDY              1
DARK               1
DEAR               1
DELIGHT            1
DESCRIBE           1
DIGNIFIED          1
DRAW               1
EMBROIDER          1
ESPECIALY          1
EVEN               1
EVERY              1
EVERYONE           1
EXCEPTIONAL        1
EXCLAIM            1
EXPRESS            1
FAMILY             1
FATHER             2
FEEL               2
FIREFLIES          1
FLOURISHES         1
FOOLISH            1
FORMING            1
FORTY              1
FROM               2
GARDEN             1
GAZING             1
GETTING            1
GOOD               1
HAPPY              1
HAS                4
HATTIFATTENERS     1
HAVE               1
HE                 2
HEARTFELT          1
HESITATION         1
HIS                1
HODGKINS           1
HOUSE              1
I                 14
IF                 2
IN                 7
INDEED             1
INSTRUCTION        1
INTO               3
IS                 3
ISN'T              1
IT                 3
LAST               1
LEAST              2
LIFE               5
LIFT               1
LIKE               2
LITTLE             1
LOOK               1
MANY               1
MATCHLESS          1
MAY                1
MEMOIR-PEN         1
MEMOIRS            1
MEMORY             1
MIND               1
MOOMIN             1
MOOMINMAMA         1
MOOMINPAPA         1
MOOMINS            1
MOOMINTROLL'S      1
MOOMINVALLEY       2
MOST               1
MUST               1
MY                11
MYSELF             1
MYSTERIOUS         1
NICE               3
NOT                3
NOTES              1
OF                15
ON                 3
ONCE               1
ONE                1
OR                 2
OVER               1
OWNER              1
PAGES              1
PAW                1
PEOPLE             1
PERISHABLE         1
PERSON             1
PERSON-AGE         1
PERSUASION         1
POISE              1
PREFACE            1
PRINT              1
QUITE              1
RATHER             1
READ               2
READERS            1
REALLY             1
REMARKABLE         1
SADNESS            1
SERIOUS            1
SHORT              1
SHOULD             2
SIGNS              1
SIMPLE             1
SITTING            1
SNOUT              1
SOME               1
SON                1
STARTING           1
STILL              1
STORMY             1
STORY              1
STRENGTH           1
SUITABLE           1
TALKING            1
TEMPTATION         1
THANKS             1
THAT               1
THE               12
THEIR              2
THINK              2
THINKS             1
THIS               4
THOUGHTFULLY       1
THREE              1
TO                 9
TONIGHT            1
TOO                1
TREMBLE            1
TRUTH              1
TRUTH-LOVING       2
VELVET             1
VERY               1
WALK               1
WANT               1
WHAT               2
WHATEVER           1
WHEN               3
WHERE              1
WHILE              1
WHO                3
WIFE               1
WILL               2
WINDOW             1
WISDOM             1
WITH               1
WORDS              1
WORK               1
WORLD              1
WRITE              2
YES                1
YIELD              1
YOU                3
YOUNG              1
YOUR               1
YOUTH              1
```

このような結果を得るには小文字を大文字に変換して処理し，アポストロフィやハイフンを単語の一部として受け付けることです．プログラムをどのように変更すればよいかは皆さんの問題にしておきましょう．この序文は約1ページでしたが出てくる単語の種類は約190種です．どんな本でも1ページに出てくる単語の種類は150～250程度のようです．配列の大きさが300あれば，大抵の本の1，2ページは処理できると思われます．32Kのメモリならばかなり余裕があるのでSIZEと文字領域を大きくすることにより，さらに多くのページ数を処理できると思います．試してみるとおもしろいですよ．

2.6　ランダムファイルに対して働くもの

ここではランダムファイルの操作に用いるステートメントや関数について説明をします．ランダムファイルがどんなものであったかを思い出しておいてください．

2．6．1　OPEN

シーケンシャルファイルの時はファイルのオープンの際にモードの指定がありましたが，ランダムファイルの場合にはモードの指定がありません．シーケンシャルファイルではファイルの先頭からの読み書き，あるいはファイルの終わりへの書き足ししかできなかったのに対し，ランダムファイルはどこからも読み書きができるので以後のファイル操作に条件を加えるようなモードの指定は無用なものです．

OPEN　〈ファイル名〉AS　#〈ファイル番号〉

指定したファイル名が存在しない場合には新しくそのファイルが作られます．

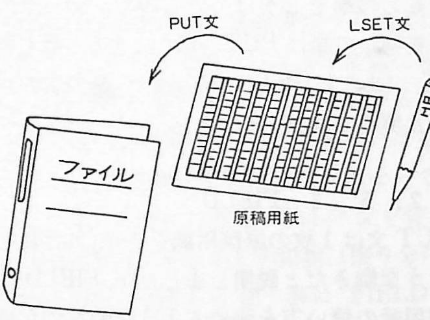
2．6．2　PUT

PUTはランダムファイルへの書き込みを行なうためのステートメントです．シーケンシャルファイルは巻物にたとえることができましたが，ランダムファイルは普通の本に相当するものです．ですから，好きなページを開いて書き込みを行なうことができます．文章を書く場合でも，「うまい序文が思い浮かばないから3章の本文から書こう．」などということをしますが，ランダムファイルでもそのような事をすることができます．PUT文はちょうど1ページ分の文章をファイルの中の指定した位置にとじ込む働きをするものです．ルーズリーフのノートや書類をとじ込むためのB5版の紙のファイルを想像していただければ良いでしょう．OPENで新しいファイル名を指定するとBASICは中身のない新品のファイルを準備して表紙に名前を書いてくれるわけです．ファイルを使う側としては，1ページ分の文章ができあがる度に，「ハイ！　これ3ページ目の文章ね！」と言って，BASICにファイルにとじ込む作業を頼むわけです．PUT文を使ってこの依頼を表わせば次のようになります．

PUT　#1，3

「1番のところにある原稿をファイルの3ページ目にとじ込んでください．」

このようにPUT文は1枚の原稿用紙をとじ込む作業能力しか無い事に注意して下さい．シーケンシャルファイルのPRINT#では1文字だけでもファイルに対して書き込むことができましたが，PUT文では常に1ページ単位のとじ込み（書き込み）しか行なえないのです．ですから，PUT文を使う前に必要な原稿用紙への書き込み作業を行なうためのLSET文が別に用意されています．



なおPUT文でとじ込む原稿用紙は256字づめということになっています．原稿用紙の使い方は次のFIELD文の説明を読んでください．

ページの指定について注意しておきます．PUT文でページ指定をする際には，できたページの分から指定をしても良いことは述べましたが，BASICはそれだけのページ数をとじこめるよう

にファイル全体の大きさを決定しますから要注意です．たとえば，PUT #1，300とすると，ファイルの300ページ目としてとじ込みを行なうのですが，BASICは1～299ページの原稿が入るようにディスク上にその領域をしっかり確保します．ですから，ファイルをすでにたくさん作ってあるディスケットを使う場合には，「300ページ目とじて下さい.」と頼んでも「1～299ページを入れる場所がないのでダメです.」と断わられます．このような時のエラーメッセージはDisk Full です．また1ページ目から順番に書いては PUT していけばよいのですから，後ろを書いては前を書いてなどというように飛び飛びに PUT して，結局何も書かないページがあったとしても，ファイル中にはそのページの領域がとられてしまうのでディスケットの記憶領域の無駄使いになります．また，このようにPUTを行なわなかったページにはどんな内容が入っているか保証されないことになるのでファイルを読むときに注意しなければなりません．ランダムファイルを作る場合でも1ページ目から順番に PUT していくのが最も確実です．BASICの方もこの事を心がけているので，ページの指定を省略して単にPUT　#1　とすると最も最近PUT（またはGET）をしたページの次のページだと解釈します．

2．6．3　FIELD

PUT文は1枚の原稿用紙をファイルに綴じ込むような働きだと説明しましたが，FIELD文は原稿用紙の使い方を決めるような働きで文章のレイアウトを決めるというのが適切なたとえでしょう．原稿用紙は256字詰という規則があるだけで縦，横の大きさは決まってませんから1次元の配列と考えることができます．FIELD文はこの配列をどのように分割して使用するかを決定します．

たとえば俳句のような詩歌は五七五というように型が決まっているので割り当てが容易に行なえます．ランダムファイルを使う場合にはいつも FIELD 文で割り当てを行ないますからこのようにある型を持った文章であることが望ましいことになります．逆に，型が頻繁に変るような場合にはその度に FIELD 文を使わねばならない事になります．実際のデータ処理では申し込み書みとか名簿というように記載事項が定まっている文書が多いので，「名前は15文字分のスペースがあれば良いだろう.」というようにして分割を行なうことができます．

具体的な FIELD 文は次のようになります．

```
FIELD　#1，5　AS A$，7 AS B$，5 AS C$
```

この例では256文字のうちの最初の5文字をA$に，次の7文字を B$，次の5文字を C$ の領域として割り当てます．ランダムファイルに対する読み書きはこのようにして FIELD 文で割り当てた文字変数を介して行ないます．原稿に書くのは文字ですから割当てのできるのは文字変数だけであることに注意してください．例では256文字のうち，割り当てを行なったのは17文字だけでしたから残りの239文字は未使用ということになります．記憶領域を有効に使うためには，割り当てを上手に行なって無駄のないようにすることも必要です．256文字を細かく割る必要がない時でも最低2つに分割することが必要となることに注意してください．

```
FIELD　#1，256 AS　A$
```

とできれば1ページ分の文字を全て A$ を介して読み書きできることになるのですが，残念ながら，文字変数の長さは255文字という制約があります．ですから，256文字を落ちこぼれなく読み書きするには最低でも2つに分けなければな

らないのです.

1ページは256文字分の文字しか入りませんから,それ以上の文字をつめ込もうとするとField over flow のエラーになります. たとえば次のような場合です.

FILD #1, 100 AS A$, 200 AS B$

FIELD 文の実際の使われ方については3章を参照してください.

2.6.4 LSET, RSET

PUT をする前の原稿用紙に対する書き込みを行なうのが LSET, RSET です. 原稿用紙256文字のうちの何文字目から書くかは, FIELD 文での指定できまります. つまり, LSET, RSET は FIELD 文で割り当てた領域への代入文に相当します.

```
FIELD #1, 20 AS A$, 20 AS B$
LSET A$= "PC-8001"
RSET B$= "N-BASIC"
```

この例では256文字のうちの最初の20文字の部分に "PC-8001" と書き, 次の20文字には "N-BASIC" と書いたことになります. ところでこの2語はどちらも20文字に満ちません. このような場合には余ったところには空白文字が入ります. LSET の場合には文字を左詰めで入れるので, PC-8001という文字の後に空白が13個入ることになります. RSET の場合には文字は右詰めで入ります. 従がって13個の空白に続いて N-BASIC と書かれることになります. 逆に書く文字の方が割り当てた領域よりも長い場合には余分な文字は無視されてしまいます. 次の例をみていただければ良くわかるでしょう.

```
10 FIELD #1,3 AS A$,3 AS B$
20 LSET A$="123456"
30 RSET B$="654321"
40 PRINT A$
50 PRINT B$
60 END
Ok
run
123
654
```

RSET の動きには注意してください. 右詰めといっても領域が足りない時には左の方から文字を入れます.

LSET, RSET で代入を行なう文字変数は, FIELD 文で領域を指定した文字変数でなければなりません. そうでないと, 正しく原稿用紙の上に書いた事になりません.

2.6.5 GET

ランダムファイルは書き込みに PUT を用いますが, 読み込みには GET を用います. PUT がファイルに対して1ページ綴じ込む仕事をするのに対し, GET はファイル中から1ページ取り出してきてくれるわけです. GET はページを取り出してくる仕事しかしてくれませんから, BASIC ではその内容を直接参照することはできません. 文字をとり扱うには文字変数に入っていなければならないからです. したがって, 読んだ内容を文字変数に一度入れなければなりませんが, この仕事は FIELD 文が行なってくれます. GET で読んだ後で FIELD 文を使えば, 読み込んだ256文字を各文字変数に振り分けてくれます. また, GET を使う前に FIELD 文を使ってあれば, その指定に従がって GET したときに振り分けてくれます.

GET 文の使い方は次のようになります.

GET #1, 3

「1番のファイルの3ページ目をとってください.」

このときに FIELD 文で次のような指示が出されていると，3ページ目の先頭の20文字がA$に，次の20文字がB$にはいることになります．

FIELD #1, 20 AS A$, 20 AS B$

GET する時にも，ページの指定に際して注意が必要です．第1に，PUT で書き込みを行なっていないページには何が書かれているかわかりません．ですから，PUT をしていないページを GET しないようにしなくてはなりません．指定したページがファイルの大きさを越える場合には Input past end のエラーになります．300ページからなる本の301ページ目をくれといっても無理な注文だからです．シーケンシャルファイルでは，ファイルの終りを知るために，EOF 関数を用いましたが，ランダムファイルでは，ファイルが何ページから成るかを知るために LOF 関数を用います．ページの指定を省略することもできますが，その場合には，その直前に PUT またはGET した次のページが対象となります．

以上の説明では，理解しやすいように，ページと呼んできましたが，この単位はレコードと呼ばれます．GET，PUT でどのレコードを扱うかの指定をすることをレコード・ナンバーの指定といい，指定を省略した時の値は，LOC関数の返す値に等しいものです．

2．6．6 LOF

LOF はファイルの最大のレコードナンバーを返す関数です．シーケンシャルファイルではファイルの終りを EOF で検出しましたが，ランダムファイルでは LOF の返す値がファイル中の最後のレコードナンバーですから，そこをファイルの終りと考えて操作することになります．

LOF 関数は次のような書式で用います．

LOF（ファイル番号）

ランダムファイルを読む時には，GET で LOF 関数の返す値よりも大きなレコードナンバーを指定してはいけません．Input past end のエラーになってしまいます．逆にファイルに対してのつぎ足しをする際には，LOF 関数の返す値よりも大きいレコードナンバーを指定して PUT を行なえば良いことになります．LOF 関数の値は，ファイルをオープンした時に決まり CLOSE するまでの間，PUT をしようが GET をしようが変わりません．LOF は，ファイルをオープンした時点でそのファイルの大きさを返す関数なのです．

2．6．7 MKI$, MKS$, MKD$

ランダムファイルに書き込む時には，文字型でなければならないことはすでに述べました．ですから数値データは，一度文字型に変換してやる必要があります．STR$ 関数を使えば次のようになります．

RSET A$=STR$ (A)

この時注意しなければならないのは，FIELD 文で指定しなければならない，A$ の領域の大きさです．変数Aが単精度実数型であれば6桁の有効数字を持ちますから，符号と小数点を含めて8文字分を準備しておかねばなりません．指数がつくときには，指数部も含めて12文字分が必要となります．このように，変数Aのとる値の範囲によってA$に必要な大きさは異なりますから，いつも12文字を与えていると領域の無駄使いになることもあるわけです．ですから，数値から，文字への変換の結果が値の範囲にかかわらず，いつも同じ長さを持つことが好ましいことになります．このような観点から，ランダムファイルへ数値を書き込む時には MKI$，MKS$，MKD$ を使って文字型への変換を行ないます．BASIC の数値データは，内部表現では，整数型

で2バイト，単精度実数型，倍精度実数型ではそれぞれ，4バイト，8バイトで表わされていますが(PC#1参照)，これらの関数は内部表現の1バイトを1文字に変換する働きをするのです．

これらの関数によって文字に変換された値を再び数値に戻すには，次に説明するCVI, CVS, CVDを使います．

2．6．8　CVI, CVS, CVD

これらの3つの関数は，それぞれ，MKI$, MKS$, MKD$によって文字に変換された数値を再び数値に戻すときに使います．次の例をみて両者の働きを理解してください．

```
10 FIELD #1,2 AS A$,4 AS B$,8 AS C$
20 A%=-32:B!=-3.25:C#=1.234567890123456#

30 LSET A$=MKI$(A%)
40 LSET B$=MKS$(B!)
50 LSET C$=MKD$(C#)
60 PRINT CVI(A$)
70 PRINT CVS(B$)
80 PRINT CVD(C$)
Ok
run
-32
-3.25
 1.234567890123456
Ok
```

CV関係の関数はMK関係の関数の逆関数になっているわけですから，正しく数値に戻すためには，MKS$を使ったならば必ずCVSを使って戻さなければなりません．また，FIELD文により文字変数の大きさを割り当てる時には，必ず必要なだけのスペースを割り当てるように注意してください．たとえば上の例でC$の大きさを5にすると，80行でのCVDによる変換の際にエラーとなってしまいます．

2．7　その他

これまでに説明してきた，ステートメントや関数は，全てファイルという構造に対して働くものでしたが，Disk　BASICの関数やステートメントの中には例外として，ファイル構造とは全く関係なしに，ディスケットそのものに対して働くものがあります．DSKI$とDSKO$の2つがそれです．この2つを使うには，フロッピーディスクに対して，正しい知識を持っていなければなりませんから，まず6章を読んでください．

2．7．1　DSKI$

DSKI$は，ディスケット上のデータを直接読みとるときに使う関数で，指定したセクタの内容からなる文字列を与えます．1セクタは256バイトですから，長さが256の文字列ということになります．

DSKI$は関数ですから普通次のような代入文の中で用います．

D$＝DSKI$（1，18，2）

1つのセクタを指定するために，DSKI$関数は，ドライブ番号，トラック番号，セクタ番号の3つの引数を必要とします．上の例では1番のディスクのトラック18，セクタ2の内容を読むことになります．ここで注意しなければならないことは，左辺のD$はダミーの変数であるということです．読み取ったデータは，0番のバッファ内に得られます．DSKI$とDSKO$では常に0番のバッファが使われるのです．そこでFIELD文を使って，0番バッファを文字変数に割りつけることによって読み取りを行ないます．GETの場合と同じことです．

```
10 FIELD ♯0，128 AS A$，128 AS B$
20 D$＝DSKI$（1，18，2）
```

この例では，A$に256バイトのうちの前半の128バイト，B$に後半の128バイトが入ります．

7章にはDSKI$の使用例がありますので参照してください．

2．7．2　DSKO$

DSKO$ は，ディスケットに対して直接書き込みをするための文です．

DSKO$　1，18，2

上の例では，1番のディスクのトラック18，セクタ2に書き込みを行ないます．書き込まれるのは0番のバッファの内容ですから，事前にLSET，RSET で書き込むデータの準備をしておかねばなりません．

DSKO$ は，ファイルの構成とは全く無関係に独立して書き込みを行なうことに注意してください．つまり，指定したセクタが，既存のファイルのデータを保持している場合にはそのデータが失なわれることになってしまいます．特にトラック18には，BASIC がファイルの管理を行なうのに必要な情報が入っていますのでこれをこわすと，そのディスケット上のデータが全て失なわれる危険すらあります．DSKO$ は通常の BASIC のプログラムを作るには不要なものですが，使用する際には，書き込むセクタの内容が変わってもかまわないかどうかを必ず確認してください．

3 「さらに詳しいオペレーション」

3．1　シーケンシャルファイル

この節は，ファイルの内容を調べるファイルダンププログラムを中心に，数値データ，文字データがディスクへどのように，記録されるか解説します．

3．1．1　ファイルダンプ

シーケンシャルファイルの説明を進めて行く上で，どのようにデータがファイル中に詰っているかを見ることができると，シーケンシャルファイルの性質がよく理解できると思います．そこで，シーケンシャルファイルの内容を，16進とASCII文字で表示するプログラムの例を示しましょう．

使い方：プログラムを正しく入力したら，RUNします．ファイルの名前を聞いてきますから，内容を見たいASCII形式のファイル名を入力します．すると，左にデータの番号が16進で表示され，中間に16進，そして，右にASCII文字が表示されます．コントロール文字は表示できないので，〝.〟に置き換えて表示します．

プログラムの説明：16進をゼロサプレスせずに表示するために，関数を2つ定義しています．FNH1$（X）関数は数値 X を16進2ケタの文字列に変換する関数であり，データの値を表示するのに用いています．FNH2$(X)は，数値Xを16進4ケタの文字列に変換する関数であり，データの番号を表示するのに用いています．シーケンシャルファイルの内容を，1文字ずつ取り出して来るのには，INPUT$関数を使っています．このようにして，EOF関数の値が真になるまで，すなわちファイルの終りまでくりかえします．このプログラムは，横40のつもりで書かれています，横80で使うには，240行の　MOD 8 を　MOD　16に，230行と250行の　TAB（30）を　TAB（54）に変更してください．

```
100 '
110 ' file dump
120 '
130 DEFFNH1$(X)=RIGHT$("0"+HEX$(X),2)
140 DEFFNH2$(X)=RIGHT$("000"+HEX$(X),4)
150 INPUT"file name";NM$
160 OPEN NM$ FOR INPUT AS #1
170 AD=0
180 A$=""
185 PRINT FNH2$(AD)" ";
190 D$=INPUT$(1,1)
200 PRINT FNH1$(ASC(D$))" ";
210 IF D$<" " THEN D$="."
220 AD=AD+1:A$=A$+D$
230 IF EOF(1) THEN PRINT TAB(30)A$:END
240 IF AD MOD 8<>0 THEN 190
250 PRINT TAB(30);A$
260 GOTO 180
```

3．1．2　数値データの入出力

まず，簡単な例で数値がどのようにファイル中に書き込まれるか実験してみましょう．

```
list
100 '
110 ' Numerical data file No.1
120 '
130 OPEN "test" FOR OUTPUT AS #1
140 FOR I=1 TO 10
150   PRINT #1,I,SQR(I)
160 NEXT
170 CLOSE #1
180 '
190 OPEN "test" FOR INPUT AS #1
200 FOR I=1 TO 10
210   INPUT #1,D1,D2
220   PRINT D1,D2
230 NEXT I
240 CLOSE #1
250 '
260 END
Ok
```

このプログラムは，1から10までの数値と，その平方根をファイルに書き込み，そのファイルの内容を読みながら，画面に表示するプログラムです．プログラムの説明をしましょう．130行でファイルをオープンします．〝test″という名前のファイルを作って，#1のバッファを通してディスクに書き込むことを宣言します．140行から160行までのFOR-NEXTループで，1から10までの数値とその平方根をファイルに書き込みます．150行のPRINT#文で，バッファ#1にとりあえず書き込まれ，もしバッファが一杯になったら，実際にディスクに書き込まれて，バッファは空になります．170行のclose文で，バッファ#1の使用終了を宣言します．このときまだバッファにデータの余りがあれば，ディスクに書き込みます．次に読み込みプログラムの説明をします．190行で，〝test″という名前のファイルを捜して，#1のバッファを通して読み出すことを宣言します．200行から230行までのFOR-NEXTループで，前に書き込んだデータを読み込みます．変数D1にIが，変数D2にSQR(I)が読み込まれます．220行で，正しく読み込まれたか画面に表示するようにしています．240行のclose文で，バッファ#1の使用終了を宣言します．実際にRUNして，プログラムが正しく働いていることを確かめてみましょう．

```
run
 1             1
 2             1.41421
 3             1.73205
 4             2
 5             2.23607
 6             2.44949
 7             2.64575
 8             2.82843
 9             3
 10            3.16228
Ok
```

正しく働いていることがわかりました．数値データのシーケンシャルファイルを使った入出力はこのように簡単にプログラムできます．

ところで，これらの数値はどのようにファイルの中に書き込まれているのでしょうか．PRINT#というステートメントによって書き込まれることから想像できるように，アスキーコードで記録されているのです．fdumpプログラムで内容を確かめてみましょう．

```
0000 20 31 20 20 20 20 20 20   1
0008 20 20 20 20 20 20 20 31          1
0010 20 0D 0A 20 32 20 20 20   .. 2
0018 20 20 20 20 20 20 20 20
0020 20 31 2E 34 31 34 32 31   1.41421
0028 20 0D 0A 20 33 20 20 20   .. 3
0030 20 20 20 20 20 20 20 20
0038 20 31 2E 37 33 32 30 35   1.73205
0040 20 0D 0A 20 34 20 20 20   .. 4
0048 20 20 20 20 20 20 20 20
0050 20 32 20 0D 0A 20 35 20   2 .. 5
0058 20 20 20 20 20 20 20 20
```

```
0060 20 20 20 32 2E 32 33 36      2.236
0068 30 37 20 0D 0A 20 36 20   07 .. 6
0070 20 20 20 20 20 20 20 20
0078 20 20 20 32 2E 34 34 39      2.449
0080 34 39 20 0D 0A 20 37 20   49 .. 7
0088 20 20 20 20 20 20 20 20
0090 20 20 20 32 2E 36 34 35      2.645
0098 37 35 20 0D 0A 20 38 20   75 .. 8
00A0 20 20 20 20 20 20 20 20
00A8 20 20 20 32 2E 38 32 38      2.828
00B0 34 33 20 0D 0A 20 39 20   43 .. 9
00B8 20 20 20 20 20 20 20 20
00C0 20 20 20 33 20 0D 0A 20      3 ..
00C8 31 30 20 20 20 20 20 20   10
00D0 20 20 20 20 20 33 2E 31        3.1
00D8 36 32 32 38 20 0D 0A      6228 ..
```

どうです，右側のアスキーダンプの部分を見れば，数値がPRINT文で画面に表示されるのと同じように記録されていることがわかります．でも，空白が目出つのはどうしたことでしょう．これは，PRINT# 文中の，変数間のカンマ〝,″に問題があるのです．

PRINT #1, I, SQR (I)

こうすると，Iを書き込んだ後，TAB (15) を行ってから，SQR (I) を書き込むのです．したがって，PRINT# 文中の変数の区切りを〝;″にすれば，余分な空白をなくすことができます．

PRINT #1, I ; SQR (I)

fdump プログラムで確かめてみましょう．

```
run"fdump"
file name? test
0000 20 31 20 20 31 20 0D 0A   1  1 ..
0008 20 32 20 20 31 2E 34 31   2  1.41
0010 34 32 31 20 0D 0A 20 33  421 .. 3
0018 20 20 31 2E 37 33 32 30     1.7320
0020 35 20 0D 0A 20 34 20 20  5 .. 4
0028 32 20 0D 0A 20 35 20 20  2 .. 5
0030 32 2E 32 33 36 30 37 20  2.23607
0038 0D 0A 20 36 20 20 32 2E  .. 6  2.
0040 34 34 39 34 39 20 0D 0A  44949 ..
0048 20 37 20 20 32 2E 36 34   7  2.64
0050 35 37 35 20 0D 0A 20 38  575 .. 8
0058 20 20 32 2E 38 32 38 34    2.8284
0060 33 20 0D 0A 20 39 20 20  3 .. 9
0068 33 20 0D 0A 20 31 30 20  3 .. 10
0070 20 33 2E 31 36 32 32 38   3.16228
0078 20 0D 0A                 ..
```

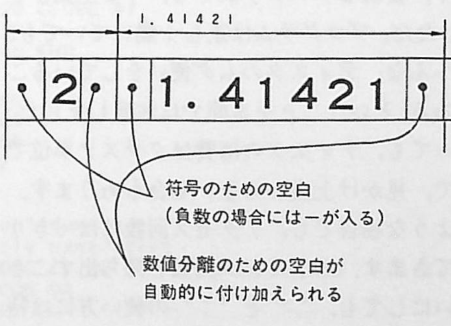
どうです，前よりずっと短かくなったことがわかります．しかし，さらに気になるのは，アスキーダンプの〝..″です，これは，左の16進ダンプを見るとわかるようにCR／LF（キャリッジリターンコードとラインフィードコード，0Dと0A）に対応しています．PRINT文で画面に出力するときに，改行するのは，この2つのコードを出力ルーチンに送っているので，残りを空白で埋めているわけではありません．画面上で連続して出力するときにそうしたように，出力並びの最後にも〝;″を付け加えればよいのです．

PRINT #1, I ; SQR (I) ;

fdump プログラムで確かめてみましょう．

```
run"fdump"
file name? test
0000 20 31 20 20 31 20 0D 0A   1  1 ..
0008 20 32 20 20 31 2E 34 31   2  1.41
0010 34 32 31 20 0D 0A 20 33  421 .. 3
0018 20 20 31 2E 37 33 32 30     1.7320
0020 35 20 0D 0A 20 34 20 20  5 .. 4
0028 32 20 0D 0A 20 35 20 20  2 .. 5
0030 32 2E 32 33 36 30 37 20  2.23607
0038 0D 0A 20 36 20 20 32 2E  .. 6  2.
0040 34 34 39 34 39 20 0D 0A  44949 ..
0048 20 37 20 20 32 2E 36 34   7  2.64
0050 35 37 35 20 0D 0A 20 38  575 .. 8
0058 20 20 32 2E 38 32 38 34    2.8284
0060 33 20 0D 0A 20 39 20 20  3 .. 9
0068 33 20 0D 0A 20 31 30 20  3 .. 10
0070 20 33 2E 31 36 32 32 38  3.16228
0078 20 0D 0A                 ..
```

どうですか，前よりさらに少し短かくなりました．まだ余白があります．なぜなら，数値の区別は空白1つでよいのに，2つの空白があります．これは数値が普通のPRINT文でどのよう



に表示されるかを考えてみれば理解できます．たとえば，2とその平方根をプリントする場合には，次のような構成になります．

したがって，この例では，数値すべて正の数ですから，符号のための空白が余分なのです．STR$ 関数を使えば，自動的に付け加えられる空白を取り除くことができます．けれども，符号のための空白が残っているので，数値を分離することができます．例えば次のようにします．

PRINT#1，STR$ (I)；STR$ (SQR (I))；

fdump プログラムで確かめてみましょう．

```
run"fdump"
file name? test
0000 20 31 20 31 20 32 20 31    1 1 2 1
0008 2E 34 31 34 32 31 20 33   .41421 3
0010 20 31 2E 37 33 32 30 35    1.73205
0018 20 34 20 32 20 35 20 32    4 2 5 2
0020 2E 32 33 36 30 37 20 36   .23607 6
0028 20 32 2E 34 34 39 34 39    2.44949
0030 20 37 20 32 2E 36 34 35    7 2.645
0038 37 35 20 38 20 32 2E 38   75 8 2.8
0040 32 38 34 33 20 39 20 33   2843 9 3
0048 20 31 30 20 33 2E 31 36    10 3.16
0050 32 32 38                  228
```

この方法は，数値すべて正の数であったから可能なのであって，負の数値の場合は，空白部分にマイナスの符号が入ってしまって，数値の分離ができなくなります．このような場合は，自動的に付け加えられる空白をそのままにしておいて，正の数値の場合は符号の空白を取り除くようにします．

数値データの読み書きは，次に述べる文字データと違って，特に注意しなければならない点は無く，安易なプログラムでも，うまく動作します．ただ，プログラムは正しく働いていても，たいへんな，ディスクのムダ使いをしていることがある，ということを本節では注意しました．といっても，ディスクの消費はクラスタ単位ですので，見かけ上変わらない場合もあります．このような場合でも，アクセス回数ははっきり違ってきます．ストリング関数を持ち出すことはないにしても，〝,〟と〝;〟の使い方には特に注意してください．

PRINT# は PRINT のようにディスクに書き込む，INPUT# は INPUT のようにディスクから読み込むと考えてもよいわけですが，数値の場合にはチョット違います．INPUT# の場合は，空白を切れ目と見なしましたが，INPUT の場合は空白は無視されるのです．

3．1．3　文字列データの入出力

文字列をシーケンシャルファイルに記録する場合の注意について述べましょう．

数値データと同じように，文字列データも PRINT 文で画面上に表示したように，シーケンシャルファイルに記録されます．文字列データで注意しなければならないのは，数値データと違って空白はデータの区切りにはならないということです．これは，文字列データの場合空白をよく使うためです．したがって，次のようなプログラムでは，思うように働きません．

```
100 '
110 ' String data file No.1
120 '
130 OPEN "test" FOR OUTPUT AS #1
140 PRINT #1,"abc","def"
150 CLOSE #1
160 '
170 OPEN "test" FOR INPUT AS #1
180 INPUT #1,A$
190 PRINT "#"A$"#"
200 INPUT #1,B$
210 PRINT "#"B$"#"
220 CLOSE #1
230 '
240 END
70 OPEN "test" FOR INPUT AS #1
180 INPUT #1,A$
190 PRINT "#"A$"#"
200 INPUT #1,B$
210 PRINT "#"B$"#"
220 CLOSE #1
230 '
240 END
Ok
```

このプログラムは，文字列〝abc〟と〝def〟を，文字変数 A$ と B$ に入力してプリントするつもりで書いたものです．それぞれの PRINT 文中の〝#〟は，文字列変数中にどのように代入

されるか,見易くするために付けておきました.

このプログラムを実行すると,

```
run
#abc         def#
Input past end in 200
Ok

run"fdump"
file name? test
0000 61 62 63 20 20 20 20 20  abc
0008 20 20 20 20 20 20 64 65        de
0010 66 0D 0A                 f..
```

とエラーを起します. 2つの文字列が空白をはさんで一緒になって A$ に読み込まれてしまい, B$ を読もうとしたときにデータがなくなってしまって, "Input past end"エラーが発生したわけです. この例ではデータが2つでしたが, 多くのデータを読み込む場合や EOF 関数を使って不定個のデータを読み込む場合などは, 一文字づつデータがずれながら読み込まれて, メチャクチャになってしまいます.

これを解決する方法は,ストリングの間に CR／LF を入れる, つまり, 2つの文字列を2つの PRINT 文で別々に書き込めばよいのです.

```
100 '
110 ' String data file No.2
120 '
130 OPEN "test" FOR OUTPUT AS #1
140 PRINT #1,"abc"
145 PRINT #1,"def"
150 CLOSE #1
160 '
170 OPEN "test" FOR INPUT AS #1
180 INPUT #1,A$
190 PRINT "#"A$"#"
200 INPUT #1,B$
210 PRINT "#"B$"#"
220 CLOSE #1
230 '
240 END

run
#abc#
#def#
Ok

run"fdump"
file name? test
0000 61 62 63 0D 0A 64 65 66  abc..def
0008 0D 0A                    ..
Ok
```

というように, うまく代入されました.

もう一つの方法は, 文字列の間に, カンマ",”を書き込む方法です.

```
100 '
110 ' String data file No.3
120 '
130 OPEN "test" FOR OUTPUT AS #1
140 PRINT #1,"abc";",";"def"
150 CLOSE #1
160 '
170 OPEN "test" FOR INPUT AS #1
180 INPUT #1,A$
190 PRINT "#"A$"#"
200 INPUT #1,B$
210 PRINT "#"B$"#"
220 CLOSE #1
230 '
240 END

run
#abc#
#def#
Ok

run"fdump"
file name? test
0000 61 62 63 2C 64 65 66 0D  abc,def.
0008 0A                       .
Ok
```

しかし, 文字列中にカンマが入っていると, 入力のときに, カンマを区切りとみなしてしまい, 文字列が分裂することになります.

```
100 '
110 ' String data file No.4
120 '
130 OPEN "test" FOR OUTPUT AS #1
140 PRINT #1,"a,c";",";"d,f"
150 CLOSE #1
160 '
170 OPEN "test" FOR INPUT AS #1
180 INPUT #1,A$
190 PRINT "#"A$"#"
200 INPUT #1,B$
210 PRINT "#"B$"#"
220 CLOSE #1
230 '
240 END

run
#a#
#c#
Ok

run"fdump"
file name? test
0000 61 2C 63 2C 64 2C 66 0D  a,c,d,f.
0008 0A                       .
Ok
```

これを解決するには，2つの方法があります．1つは，ディスクに書く時に文字列をクォーテーションマークで囲んでおく方法です．

```
list
100 '
110 ' String data file No.5
120 '
130 OPEN "test" FOR OUTPUT AS #1
135 Q$=CHR$(34)
140 PRINT #1,Q$;"a,c";Q$;Q$;"d,f";Q$
150 CLOSE #1
160 '
170 OPEN "test" FOR INPUT AS #1
180 INPUT #1,A$
190 PRINT "#"A$"#"
200 INPUT #1,B$
210 PRINT "#"B$"#"
220 CLOSE #1
230 '
240 END
Ok

run
#a,c#
#d,f#
Ok

run"fdump"
file name? test
0000 22 61 2C 63 22 22 64 2C   "a,c""d,
0008 66 22 0D 0A               f"..
Ok
```

クォーテーションマークは直接書くことができないので，文字変数 Q$ にクォーテーションマークを代入して使っています．クォーテーションマークの ASCII コードは34です．

もう一つの方法は，入力に LINE INPUT# 文を使う方法です．LINE INPUT#文は，CR／LF コードが来るまで，読み込みます．

```
list
100 '
110 ' String data file No.6
120 '
130 OPEN "test" FOR OUTPUT AS #1
140 PRINT #1,"a,c"
145 PRINT #1,"d,f"
150 CLOSE #1
160 '
170 OPEN "test" FOR INPUT AS #1
180 LINE INPUT #1,A$
190 PRINT "#"A$"#"
200 LINE INPUT #1,B$
210 PRINT "#"B$"#"
220 CLOSE #1
230 '
240 END
Ok
```

```
run
#a,c#
#d,f#
Ok

run"fdump"
file name? test
0000 61 2C 63 0D 0A 64 2C 66   a,c..d,f
0008 0D 0A                     ..
Ok
```

特殊な場合として，英語の文章などがあります．実際の文章には，CR／LF コード，引用符，カンマなどが複雑に表われるので別の方法を考えなければなりません．これには，INPUT$文で1文字ずつディスクから読まなければなりません．どのように読むかというと，1つは，ディスクに書き込むときに，ストリングの間にその文章には出てこない文字を入れておく方法です．例えば，0FF (16進) などは，どの文字にも対応しませんから，使うことができます．

もう一つの方法は，ディスクに書き込むときに，一諸にそのストリングの長さを数値データとしておく方法です．この方法を使えば，特殊な文字を使わなくてもよいのです．

3．2　ランダムファイル

この節は，レコード番号によるアクセスからはじまり，名前によるアクセス，そのとき用いられる対応表の扱い，さらに未使用となったレコードの再利用など，ランダムファイルを使う上で，基本的なテクニックを説明します．

3．2．1　レコード番号によるアクセス

最も簡単なランダムファイルの使用法は，レコード番号によるアクセスです．登録番号，商品番号など，数値によってデータを引き出す機会は多いと思います．後で述べる名前による呼出しにくらべて，登録番号とレコード番号が1対1に対応する場合はもちろん，その他の場合でもレコード番号との対応づけが容易であるのが特徴です．その一例を示します．

```
100 '
110 ' Example of access by number
120 '
130 KILL "test"
140 '
150 OPEN "test" AS #1
160 FIELD #1,34 AS D$
170 '
180 PRINT
190 INPUT "write or read";A$
200 IF A$="end" THEN 330
210 INPUT "number";N
220 IF LEFT$(A$,1)="w" THEN 280
230 '
240 GET #1,N
250 PRINT "data:";D$
260 GOTO 180
270 '
280 INPUT "data:";D1$
290 LSET D$=D1$
300 PUT #1,N
310 GOTO 180
320 '
330 CLOSE #1
340 END
```

130行で同名のファイルを KILL しておきます．これは，後でファイルの成長を確かめるためです．150行でファイルをオープンします．160行で，文字変数 D$ にバッファを割り当てます．34文字を与えてありますが，特に意味があるわけではありません．190行で，ファイルに書き込むのか，読み込むのか聞いてきます．200行で〝end″が入力されたら，終りになるようにしています．210行で，レコード番号を入力します．入力されたコマンドが〝w″で始まれば，書き込み，それ以外は読み出しとみなされます．240行からが読み出し，280行からが書き込みです．文字変数 D1$ は，INPUT 文によって直接バッファに書き込めないので，LSET 文によって代入するための仮変数です．INPUT 文で D$ を使うと，FIELD 文の効果が無くなります．

このプログラムを走らせてみます．

```
run

write or read? w
number? 1
data:? data1

write or read? w
number? 2
data:? data2
```

```
write or read? r
number? 1
data:data1

write or read? r
number? 2
data:data2

write or read? r
number? 3
Input past end in 240
Ok
```

1番，2番と書き込んだ後，1番，2番と読み込んで表示します．1番レコードに書き込んだ内容は〝data1″，2番レコードに書き込んだ内容は〝data2″という，短いものです．さらに，3番レコードを読もうとすれば，〝Input past end″エラー，すなわち，3番レコードは書いてないので存在しない，ファイルの長さは2レコードとなります．

シーケンシャルファイルは，連続的にしか成長しませんが，ランダムアクセスファイルは，レコード番号の指定の仕方によっては，不連続に成長することがあります．もう一度，先程のプログラムを走らせてみましょう．

```
write or read? r
number? 9
data:data9

write or read? r
number? 8
data:

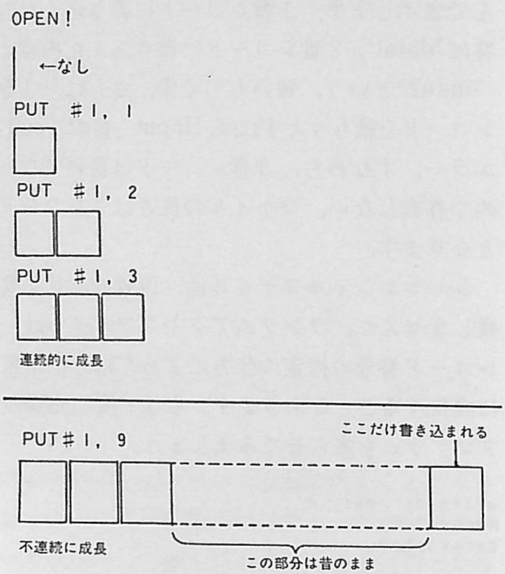
write or read? r
number? 5
data:

write or read? r
number? 10
Input past end in 220
Ok
```

この例でわかるように，間を飛ばしていきなり9番レコードに書き込めば，間のレコードに特に書き込まなくても，ファイルの長さは9レコードになります．したがって，間のレコードを読み出そうとすれば，前に書いてあったゴミのようなデータが読み出されることがあります．

シーケンシャルファイルでは，ファイル全体に有効データが詰まっていますが，ランダムフ

ァイルでは，PUT# しないレコードの内容は昔のままで，特に定まった内容を含んでいません．したがって，ランダムファイルに初めからデータを詰めて書き込むようにするためには，PUT# するレコード番号を変数として，PUT# するたびに1つずつ増していくようにするか，または，連続して PUT# する場合にはレコード番号を省略すれば，自動的にインクリメントしながら PUT# することができます．



このようにレコード番号を不連続に使用すると，ランダムファイルも不連続に拡大します．

3．2．2　ランダムファイルへの追加

シーケンシャルファイルに新しい情報を追加するには，ファイルをオープンするときに，モードを　for　append とすることで可能でした．ランダムファイルの場合はどのようにすればよいのでしょうか．前にも説明したように，新しいレコードを追加するためには，前に作ったファイルの最大レコード数を知る必要があります．それを知らせてくれる関数が LOF 関数です．ファイルを初めて作ったときの LOF 関数の値は0です．この関数を使うときに注意しなければならないのは，前に作ったファイルの最大レコード番号を与えるのであって，現在のファイルの最大レコード番号を与えるのではないのです．つまり，その値はファイルをオープンしたときに決まり，ファイルをクローズするまでその値は変化しません．したがって，LOF 関数を正しく使うには，例えば次のように使います．

```
100 '
110 ' Example of access by number
120 '
130 OPEN "test" AS #1
140 FIELD #1,34 AS D$
150 MAX=LOF(1)
160 PRINT "MAX=";MAX
170 '
180 PRINT
190 INPUT "write or read";A$
200 IF A$="end" THEN 350
210 IF LEFT$(A$,1)="w" THEN 280
220 '
230 INPUT "number";N
240 GET #1,N
250 PRINT "data:";D$
260 GOTO 180
270 '
280 MAX=MAX+1
290 PRINT "number:";MAX
300 INPUT "data:";D1$
310 LSET D$=D1$
320 PUT #1,MAX
330 GOTO 180
340 '
350 CLOSE #1
360 END
```

```
MAX= 0

write or read? w
number: 1
data:? data1

write or read? w
number: 2
data:? data2

write or read? w
number: 3
data:? data3

write or read? r
number? 1
data:data1

write or read? r
number? 2
data:data2

write or read? end
Ok
```

```
run
MAX= 3

write or read? r
number? 1
data:data1

write or read? w
number: 4
data:? data4

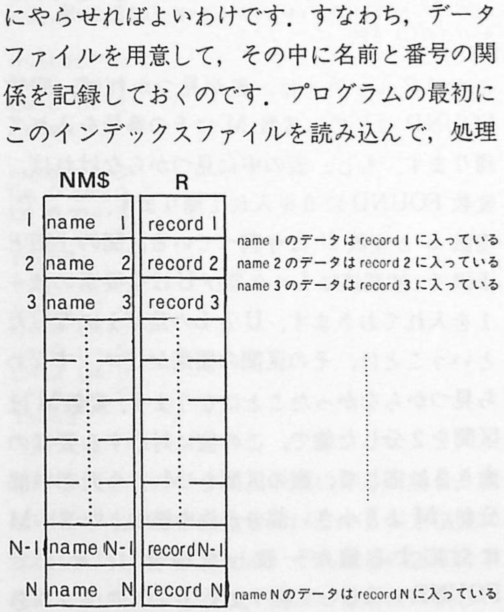
write or read? w
number: 5
data:? data5

write or read?
Break in 190
Ok
```

この例で，変数 MAX は現在の最大レコード番号で，新しくレコードを追加するには，MAX＋1番目のレコードを PUT すればよいわけです．

3．2．3 対応表によるアクセス

前に述べたレコード番号によるアクセスは，登録番号や商品番号による問合せには，便利ですが，名前や商品名によって，情報を取り出すには不便です．なぜなら，名前と番号との関係をあらかじめ知っていなければならないからです．このようなことは，もちろんコンピュータにやらせればよいわけです．すなわち，データファイルを用意して，その中に名前と番号の関係を記録しておくのです．プログラムの最初にこのインデックスファイルを読み込んで，処理



中はメモリ中の対応表を見ながら処理を進め，変更もメモリ中の対応表に対して行ない，処理が終ったら，インデックスファイルとして再びディスクに書き込んでおけばよいのです．インデックスファイルは，始めにまとめて読み出し，終りにまとめて書き込むのですから，シーケンシャルファイルを使えばよいでしょう．

次に対応表をどのように引くか説明します．まず考えられる方法は，前の方から順々に比較していく方法です．

例えば，

```
100 '
110 ' sequencial search
120 '
130 NM$(N+1)=KY$
140 I=1
150 IF NM$(I)=KY$ THEN 180
160 I=I+1
170 GOTO 150
180 IF I=N+1 THEN FOUND=0 ELSE FOUND=-1
190 RETURN
```

文字配列 NM$（I）は，対応表の名前の部分です．変数 N は，名前の総数，文字変数 KY$ は，探している名前です．このサブルーチンは NM$（N＋1）に KY$ を入れて番兵としています．こうすることによって，ループの中で I と N を比較する必要がなくなります．ループが終ったときに，I の値が N＋1 ならば，対応表の中に，KY$ は載っていなかったことになり，そうでなければ，KY$ が載っており，その配列の番号が I に入っています．したがって，R（I）の中に求めるレコード番号が入っているのです．見つかった場合には，変数 FOUND に－1を代入して，見つからなかった場合には，0を代入して戻ります．

例えば，

```
100 '
110 ' sequencial search
120 '
130 FOR I=1 TO N
140    IF NM$(I)=KY$ THEN FOUND=-1:RETURN
150 NEXT I
160 FOUND=0
170 RETURN
```

このサブルーチンでは，FOR-NEXT ループを使って，ループの終了条件を判定しています．

以上のように順々に比較していく方法は，要素の数が少ないときは，簡単な方法なのですが，要素の数が大きくなると，平均の手間は要素の数に比列してかかるようになります．もっとうまい方法はないでしょうか．それは，みなさんが辞書を引くときのことを考えれば，思い当るはずです．調べたい単語を見つけるために辞書の第一ページから見ていきますか，そんなバカなことはしていないはずです．まずおよその目やすをつけます．たとえば，「basic」を調べたいなら，〝b〟の部分を開きます．そして〝ba〟で始まる単語，〝bas〟で始まる単語というように，除々に単語の数を減らしていって〝basic〟にたどりつくわけです．このようなことができるのは，なぜでしょうか．それは，辞書の中の単語がアルファベット順になっているからです．〝basic〟は〝b〟で始まる部分に載っていて〝a〟や〝c〟で始まる単語の部分には載っていない．〝a〟より後で〝c〟より前に載っているはずだ．〝ba〟は〝bb〟より前，〝bas〟は〝bar〟より後で，〝bat〟より前だ，このようにして調べる必要のない所は，調べずに目的の所へ行き着くことができるわけです．この方法を対応表に使ってみましょう．まず，対応表をアルファベット順にします．もちろん新しく作るのだったら話は簡単ですし，すでに対応表がある場合でも PC#1 で述べたような，ソーティングプログラムによって，アルファベット順に並べ直すことができます．正確に言うと，アルファベット順というよりは，ASCII コード順と言った方がよいかもしれません．これは，アルファベットですとアルファベット順，カタカナですと，小文字の一部を除いてアイウエオ順です．別にこんなことはどうでもよくて，単語の大小関係が一意的に決定できればよいわけです．こうして順序よく並べられた表を引くのに，2分検索(binary search)がよく使われます．これは，ハイとイイエで目的を見つけるものです．例えば，次のような場合です．

A	B	C	D	E	F	G

この中からFを見つけます，まず，真中のDと比較します．DよりFの方が大きいので右半分にあることになります．

E	F	G

真中のFと比較します．これで見つかりました．つまり，2回の比較で見つけられたわけです．もし，順序よく比較していったらどうなるでしょう．AとBとCとDとEとFと，6回も比較しなければなりません．一般に2分検索での手間は $\log_2$(要素の数) に比列します．したがって，要素の数が多くなればなるほど，大きな差が出てくるのです．そのプログラムの一例を示します．

```
410 '
420 ' binary search
430 '
440 L=0:U=N+1
450 IF U-L=1 THEN FOUND=0:RETURN
460 M=(U+L)¥2
470 IF NM$(M)=KY$ THEN FOUND=-1:RETURN
480 IF NM$(M)<KY$ THEN L=M ELSE U=M
490 GOTO 450
```

このサブルーチンは，表が見つかれば，変数 FOUND にして，変数Mにその番号を入れて帰ります．もし，表の中に見つからなければ，変数 FOUND に0を入れて帰ります．ここで，変数Lと変数Uは今調べている区間の上限と下限で，初期値はL＝0及びUは全要素の数＋1を入れておきます．UとLの差が1になったということは，その区間の要素がゼロ，すなわち見つからなかったことになります．変数Mは区間を2分した値で，この値に対応する要素の大きさに応じて，次の区間をMより大きい部分か，Mより小さい部分か決めます．ズバリMに対応する値が一致した場合は，そこで FOUND＝-1として戻ります．次のプログラムを

走らせて見ると，２分検索の様子がよく理解できると思います．このプログラムでは，表の要素は１から10までの数字です，サブルーチンに，L，U，M のそれぞれの値を表示する行を加えてあります．

```
100 '
110 ' Binary search demo
120 '
130 N=9
140 FOR I=1 TO N
150   NM$(I)=RIGHT$(STR$(I),1)
160 NEXT I
170 '
180 INPUT "number";KY$
190 GOSUB 250
200 IF FOUND THEN PRINT "found"
210 IF NOT FOUND THEN PRINT "not found"
220 PRINT
230 GOTO 180
240 '
250 ' binary search
260 '
270 L=0:U=N+1
280 PRINT USING "L=## U=##";L,U
290 IF U-L=1 THEN FOUND=0:RETURN
300 I=(U+L)¥2
310 IF NM$(I)=KY$ THEN FOUND=-1:RETURN
320 IF NM$(I)<KY$ THEN L=I ELSE U=I
330 GOTO 280

run
number? 1
L= 0 U=10
L= 0 U= 5
L= 0 U= 2
found

number? 2
L= 0 U=10
L= 0 U= 5
found

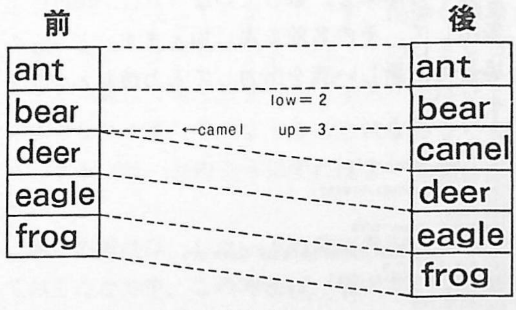
number? 3
L= 0 U=10
L= 0 U= 5
L= 2 U= 5
found

number? 4
L= 0 U=10
L= 0 U= 5
L= 2 U= 5
L= 3 U= 5
found
```

対応表をアルファベット順に作った場合，新しい名前を登録するときに，しかるべき位置に入れなければなりません．それはどのようにすればよいのでしょうか．表の一番最後に新しい名前を付け足して，要素が一つ増たことにして，ソーティングサブルーチンを作って，アルファベット順に並べる．これは，ソーティングプログラムによっては時間がかかるし，その名前を付け加える前は完全にアルファベット順であったのですから，なんとかならないものでしょうか．それは２分検索プログラムを使えば簡単なのです．検索プログラムは，名前が見つからなかった場合にも，有用なデータを与えます．すなわち，名前が表に載ってなかった場合，新しい名前のしかるべき位置は，L と U の間であるはずです．したがって，U から後ろの全要素を後へ１つずつずらし，U の位置に新しい名前を入れればよいわけです．例えば次のようなプログラムになります．

```
220 '
230 ' insert
240 '
250 PRINT "insert"
260 FOR I=N TO U STEP -1
270   NM$(I+1)=NM$(I):R(I+1)=R(I)
280 NEXT I
290 N=N+1:MAX=MAX+1
300 NM$(U)=KY$:R(U)=MAX
310 RETURN
```

対応表への挿入



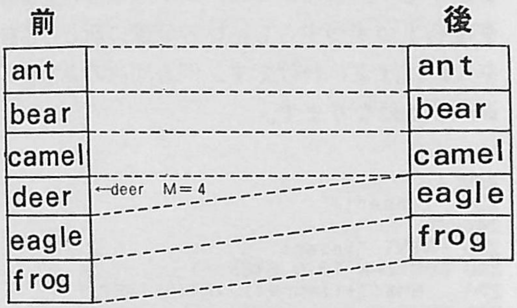
対応表にｃａｍｅｌを挿入する

すでに登録されている単語を削除するには，その要素をヌルストリングにするのではダメで，そこをつめておかなければなりません．なぜなら，ヌルストリングは最も小さい文字列要素ですから，表の途中で表われては２分検索がうま

くいきません．その方法は，前の挿入と似ています．その要素から後ろをすべて，前へ一つづつずらせばよいのです．例えば，次のようなプログラムになります．

```
320 '
330 ' delete
340 '
350 PRINT "delete"
360 N=N-1
370 FOR I=M TO N
380  NM$(I)=NM$(I+1):R(I)=R(I+1)
390 NEXT I
400 RETURN
```

対応表からの削除



対応表からｄｅｅｒを削除する

次のプログラムは，入力された文字列が表に載っていれば，〝delete〞と表示して，その名前を表から削除し，載っていなければ〝insert〞と表示して，その名前を表に加えます．どちらの場合も，新しい表を出力して入力待ちとなります．

```
100 CLEAR 1000
110 SIZE=100
120 DIM NM$(SIZE),R(SIZE)
130 '
140 PRINT
150 INPUT "name";KY$
160 GOSUB 420
170 IF FOUND THEN GOSUB 330 ELSE GOSUB 230
180 FOR I=1 TO N
190   PRINT NM$(I),R(I)
200 NEXT I
210 GOTO 140
220 '
230 ' insert
240 '
250 PRINT "insert"
260 FOR I=N TO U STEP -1
270   NM$(I+1)=NM$(I):R(I+1)=R(I)
280 NEXT I
290 N=N+1:MAX=MAX+1
300 NM$(U)=KY$:R(U)=MAX
310 RETURN
320 '
330 ' delete
340 '
350 PRINT "delete"
360 N=N-1
370 FOR I=M TO N
380  NM$(I)=NM$(I+1):R(I)=R(I+1)
390 NEXT I
400 RETURN
410 '
420 ' binary search
430 '
440 L=0:U=N+1
450 IF U-L=1 THEN FOUND=0:RETURN
460 M=(U+L)¥2
470 IF NM$(M)=KY$ THEN FOUND=-1:RETURN
480 IF NM$(M)<KY$ THEN L=M ELSE U=M
490 GOTO 450
```

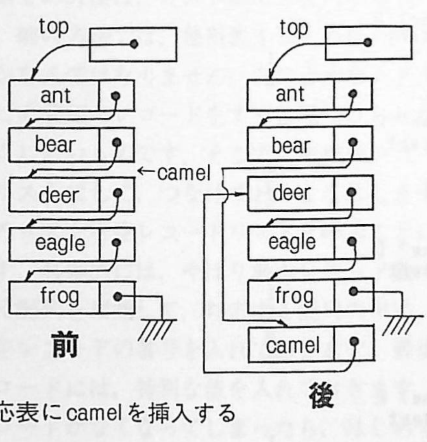
以上のような構造は，単純でしかも検索が速いのが特徴です．しかし，挿入と削除では，データそのものを移動するので時間がかかります．したがってこのような構造は，後で紹介する「辞書」プログラムに使われています．辞書は引くものですから，検索は速くなければならないのですが，挿入は初めにデータを入力するとき以外は，ほとんど行なわれませんし，削除することは，ほとんどないからです．

検索が速く，挿入削除が容易である，そんな条件を満してくれるのが，二分木構造です．これは，2章に例が載っているのでそちらを参照してください．

挿入と削除を容易にするには，リスト構造があります．リスト構造は，配列中の位置ではなくて，その名前に対応するポインタによって順序が決っています．したがって，配列の第一要素が最も小さな要素とはかぎりません，最も小さな要素は，ポインタ変数 TOP によって示されるのです．TOP によって示される要素のポインタを調べると，次の要素を指しています　そして，その要素のポインタを調べるとその次の要素を指しているというように，イモズル式に要素をたどることができるのです．最後の要素に対応するポインタには，最後を示す特別な値を入れておきます．例えば，配列を添字1から使用すれば，ゼロが最後を示すように決めることができます．図ではこれを，アース記号で表現しています．図で説明すれば，TOP に入っている値は1，この場合は配列の第一要素とリストの第一要素が一致しています．そして，ant のポインタには2，すなわち bear を指していま

す．同様にして bean のポインタには 3 が，deer のポインタには 4 が，eagle のポインタには 5 が，そして，frog のポインタには 0 が，入っているのです．リスト構造の場合新しい要素の挿入は簡単で，例えば，camel を挿入する場合，camel 自身は配列の後に付きますが，bear の deer を指している矢印を切って，すなわち，bear のポインタを camel に，camel のポインタを deer にすればよいのです．要素の削除も簡単で，削除したい要素を指しているポインタを，削除したい要素の指している値に書き変えればよいのです．例えば，deer を削除する場合は，deer を指している要素すなわち camel のポインタの値を，deer が指している要素，すなわち eagle に書き変えればよいのです．挿入，削除で注目してほしいのは，要素自身は配列中をまったく移動しないで，ポインタを書き変えているだけだということです．これは，前の配列構造と大きく異なっています，前の構造では配列の前の方で挿入，削除が起ったら，たいへんな数の要素を移動しなければなりません．しかし，リスト構造の欠点は，二分検索が使えないことです．

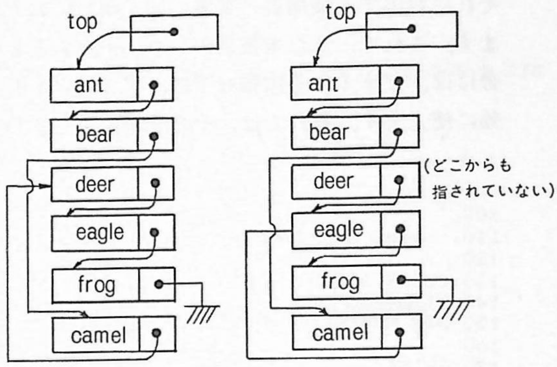
リスト構造による挿入



対応表に camel を挿入する

したがって，逐次検索を用いなければならず，検索は遅くなります．

リスト構造による削除



対応表から dear を削除する

3．2．4　ゴミレコードの再利用

今までの説明で，対応表の使い方は理解できたと思います．削除が行なわれなければ，レコード番号の若い順に PUT していくので未使用レコードは発生しません．ところが，削除が行なわれると，その名前に対応したレコードは，どこからも指されないレコード，すなわち，ゴミレコードになってしまうのです．削除が頻繁に起こるようなプログラムでは，このゴミレコードが大量にできてしまい，ディスク上はゴミの山となります．そのようなことにならないようにするには，どのようにすればよいのでしょうか．

第一の方法は，レコードの使用状況を記録しておくことです．この方法は，最大レコード数の配列（使用表とでも呼んでおきます）を用意して，使用したレコードには，対応する配列要素に，ある数を入れて印を付けておきます．もちろん逆に，使用していないレコードに印を付けておいてもかまいません．新しくレコードを使用するときは，この使用表を調べて，もし未使用レコードがあったら，そのレコードを使用

し，なかったら，はじめて新しいレコードをファイルの最後部に付け加えます．そのときには，使用表の要素を一つ増して，印を付けておきます．削除によって，未使用となったレコードは，それに対応する使用表の要素の印を消しておきます．これで，まだ未使用のレコードがある場合には，ファイルを拡張せずに，デイスクを有効に使えます．それでは，その一例を示しましょう．

```
100 '
110 ' used map demo
120 '
130 CLEAR 1000
140 SIZE=100
150 DIM NM$(SIZE),R(SIZE),USED(SIZE)
160 '
170 PRINT
180 INPUT "name";KY$
190 GOSUB 530
200 IF FOUND THEN GOSUB 430 ELSE GOSUB
    270
210 IF N=0 THEN 250
220 FOR I=1 TO N
230   PRINT NM$(I),R(I)
240 NEXT I
250 GOTO 170
260 '
270 ' insert
280 '
290 PRINT "insert"
300 FOR I=N TO U STEP -1
310   NM$(I+1)=NM$(I):R(I+1)=R(I)
320 NEXT I
330 N=N+1
340 NM$(U)=KY$
350 '
360 I=1
370 IF USED(I) THEN I=I+1 :GOTO 370
380 IF I=MAX+1 THEN MAX=MAX+1
390 USED(I)=-1
400 R(U)=I
410 RETURN
420 '
430 ' delete
440 '
450 PRINT "delete"
460 N=N-1
470 USED(R(M))=0
480 FOR I=M TO N
490   NM$(I)=NM$(I+1):R(I)=R(I+1)
500 NEXT I
510 RETURN
520 '
530 ' binary search
540 '
550 L=0:U=N+1
560 IF U-L=1 THEN FOUND=0:RETURN
570 M=(U+L)¥2
580 IF NM$(M)=KY$ THEN FOUND=-1:RETURN
590 IF NM$(M)<KY$ THEN L=M ELSE U=M
600 GOTO 560
```

このプログラムは，対応表の説明で使用したものに，ゴミレコード処理を加えたものです．配列 USED は，使用表で，使用中のレコード番号に対応する要素には－1が未使用のレコード番号に対応する要素には0が入ります．挿入サブルーチンに新しく加わったのが，360行から390行です．使用表を1っ1っ調べながら要素が0である番号を探します．このループには上限の判断がありませんが，配列を宣言するときに，すべての要素に0が代入されるので，すなわち，すべて未使用状態になっているので，そこでストップします．そして，未使用のレコード番号が，変数Ｉに入ってループを終ります．変数Ｉが最大レコード番号＋1であったら，未使用レコードが存在しなかったことになりますから，最大レコード番号を更新します．使用したレコードに対応する，使用表の要素には－1を代入しておきます．削除サブルーチンに新しく加わったのは，470行だけで，未使用になったレコードに対応する要素に0を代入します．次に，このプログラムを走らせた結果を示します．

```
name? A
insert
A              1

name? B
insert
A              1
B              2

name? C
insert
A              1
B              2
C              3

name? D
insert
A              1
B              2
C              3
D              4

name? E
insert
A              1
B              2
C              3
```

```
D              4
E              5

name? C
delete
A              1
B              2
D              4
E              5

name? D
delete
A              1
B              2
E              5

name? F
insert
A              1
B              2
E              5
F              3

name? G
insert
A              1
B              2
E              5
F              3
G              4
```

名前を A, B, C, D, E と順番に登録して，それぞれ，1，2，3，4．5のレコード番号が与えられました．次に，CとDを削除して，3と4が未使用レコードになりました．次にFとGを登録しますが，未使用レコードがあるので，そのレコード番号が与えられました．

第2の方法は，リスト構造を使用するものです．前の方法では，使用表をはじめから調べて行かなくてはなりません．このようなムダをせずに未使用のレコードをすぐに見つけられるようにしたいものです．そこで，未使用レコードをリストにして，つなげておくようにします．このリストを空レコードリストと呼ぶことにします．具体的には，やはり最大レコード数の一次元配列を用意して，対応する配列の中に，次の空レコードの番号を入れておきます．最後のレコードには，特別な値を入れておきます．空レコードがなくなってしまったら，はじめて新しいレコードを，ファイルの最後部に付けます．

```
100 '
110 ' free list demo
120 '
130 CLEAR 1000
140 SIZE=100
150 DIM NM$(SIZE),R(SIZE),FL(SIZE)
160 '
170 PRINT
180 INPUT "name";KY$
190 GOSUB 490
200 IF FOUND THEN GOSUB 390 ELSE GOSUB 270
210 IF N=0 THEN 250
220 FOR I=1 TO N
230   PRINT NM$(I),R(I)
240 NEXT I
250 GOTO 170
260 '
270 ' insert
280 '
290 PRINT "insert"
300 FOR I=N TO U STEP -1
310   NM$(I+1)=NM$(I):R(I+1)=R(I)
320 NEXT I
330 N=N+1
340 NM$(U)=KY$
350 '
360 IF FR THEN R(U)=FR: FR=FL(FR)
         ELSE MAX=MAX+1: R(U)=MAX
370 RETURN
380 '
390 ' delete
400 '
410 PRINT "delete"
420 N=N-1
430 FL(R(M))=FR: FR=R(M)
440 FOR I=M TO N
450   NM$(I)=NM$(I+1):R(I)=R(I+1)
460 NEXT I
470 RETURN
480 '
490 ' binary search
500 '
510 L=0:U=N+1
520 IF U-L=1 THEN FOUND=0:RETURN
530 M=(U+L)¥2
540 IF NM$(M)=KY$ THEN FOUND=-1:RETURN
550 IF NM$(M)<KY$ THEN L=M ELSE U=M
560 GOTO 520
```

リストを記憶する配列は FL です．挿入サブルーチンの追加は，360行です．変数 FR は，未使用レコードリストの先頭を指しているポインタで，その値が0であるときには，未使用レコードが存在しないことを示しています．削除サブルーチンの追加は，430行です．未使用となったレコード番号 R (M) を未使用リストの先頭に加えます．

次に実行結果を示します．

```
name? A
insert
A                   1

name? B
insert
A                   1
B                   2

name? C
insert
A                   1
B                   2
C                   3

name? D
insert
A                   1
B                   2
C                   3
D                   4

name? E
insert
A                   1
B                   2
C                   3
D                   4
E                   5

name? C
delete
A                   1
B                   2
D                   4
E                   5

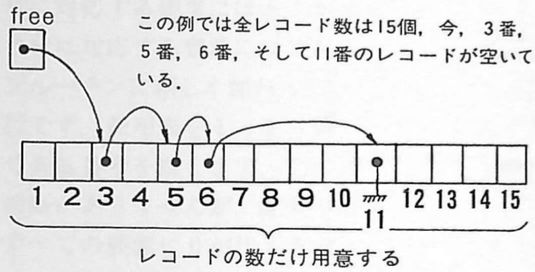
name? D
delete
A                   1
B                   2
E                   5

name? F
insert
A                   1
B                   2
E                   5
F                   4

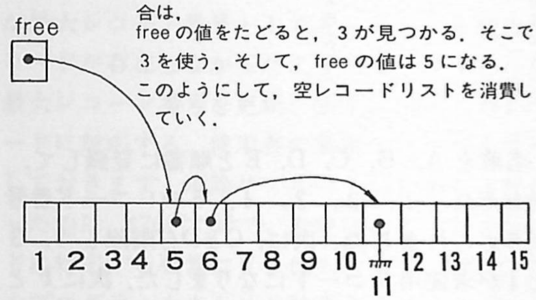
name? G
insert
A                   1
B                   2
E                   5
F                   4
G                   3
```

前と同じように，A～E を登録したあとに，C と D を削除し，その後に F と G を新たに登録しています．ここで注意してほしいのは，F と G に割り当てられたレコード番号です．前のプログラムでは，F に 3 番，G に 4 番が割り当てられましたが，今度はその逆になっています．これは，リストを作る手順を考えてみれば理解できます．

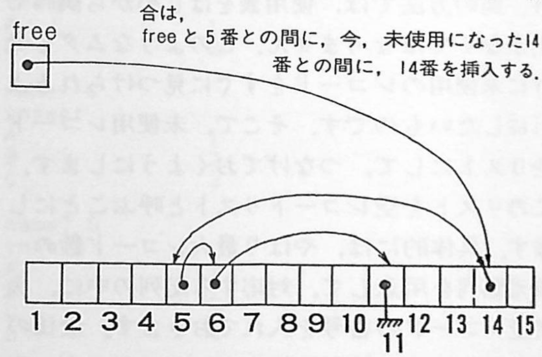
空レコードリスト



この例では全レコード数は15個，今，3番，5番，6番，そして11番のレコードが空いている．



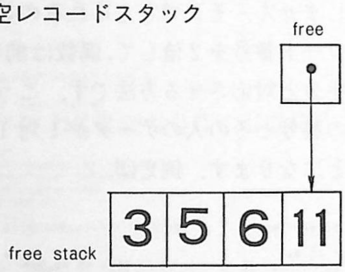
前の状態から，新しくレコードを使いたい場合は，
freeの値をたどると，3が見つかる．そこで3を使う．そして，freeの値は5になる．
このようにして，空レコードリストを消費していく．



前の状態から，未使用レコードが発生した場合は，
freeと5番との間に，今，未使用になった14番との間に，14番を挿入する．

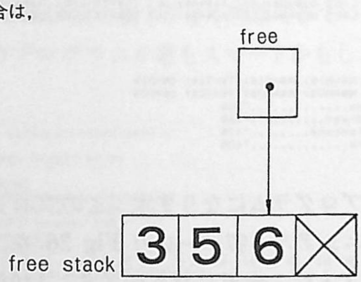
第3の方法は，スタック構造を使うものです．前の2つの方法ではいづれも，レコード数だけの配列を用意しなければなりませんでした．後で述べるような，レコード間のリンク(つながり)がある場合には，配列を共用できますが，ない場合には，特に全レコード数が多くて，未使用レコードが少ない場合には，ムダが多くなります．そんな場合には，未使用レコードのみを保存する配列を用意しておけばよいのです．もちろん，配列に必要な記憶領域は，始めに確保しておかなければなりません，したがって，実際に主記憶を占有する量は変わらない場合があります，といっても，全部消去するなら，作り直した方がましなのです．

空レコードスタック



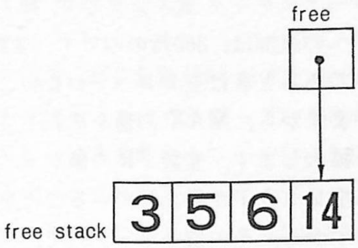
この例では，前と同じ様に，3・5・6・11番のレコードが空いている．freeの値は，今4が入っている．

前の状態から，新しくレコードを使いたい場合は，



この例では，11番のレコードが，使用されることになったことを示している．freeの値は，今，3が入っている．

前の状態から，未使用レコードが発生した場合は，



この例では，14番のレコードが，未使用レコードになったことを示している．FREEの値は，今，14が入っている．

次にプログラムを示します．

```
100 '
110 ' free stack demo
120 '
130 CLEAR 1000
140 SIZE=100
150 DIM NM$(SIZE),R(SIZE),FS(SIZE)
160 '
170 PRINT
180 INPUT "name";KY$
190 GOSUB 490
200 IF FOUND THEN GOSUB 390 ELSE GOSUB 270
210 IF N=0 THEN 250
220 FOR I=1 TO N
230   PRINT NM$(I),R(I)
240 NEXT I
250 GOTO 170
260 '
270 ' insert
280 '
290 PRINT "insert"
300 FOR I=N TO U STEP -1
310   NM$(I+1)=NM$(I):R(I+1)=R(I)
320 NEXT I
330 N=N+1
340 NM$(U)=KY$
350 '
360 IF FR THEN R(U)=FS(FR): FR=FR-1
        ELSE MAX=MAX+1: R(U)=MAX
370 RETURN
380 '
390 ' delete
400 '
410 PRINT "delete"
420 N=N-1
430 FR=FR+1: FS(FR)=R(M)
440 FOR I=M TO N
450   NM$(I)=NM$(I+1):R(I)=R(I+1)
460 NEXT I
470 RETURN
480 '
490 ' binary search
500 '
510 L=0:U=N+1
520 IF U-L=1 THEN FOUND=0:RETURN
530 M=(U+L)¥2
540 IF NM$(M)=KY$ THEN FOUND=-1:RETURN
550 IF NM$(M)<KY$ THEN L=M ELSE U=M
560 GOTO 520
```

配列 FS が空レコードスタックで，変数 FR が空レコードスタック ポインタです．挿入サブルーチンへの追加は，360行のみです．変数 FR の値が0であるときは空レコードがないことを示していますから，MAX の値を更新して，ファイルを拡大します．変数 FR の値が0でないときは，空レコードが存在していることを，示していますから，その値をポップして，スタックポインタの値を一つ減らします．削除への追加は，430行のみです．スタックポインタの値を1つ増して，空になったレコード番号をスタックに積みます．これを実行すると，空リストの場合と同様な結果が得られます．

3．2．5　レコードの分割

ランダムアクセスファイルの最大の目的は，高速の呼び出しです．そのために，固定長レコード（マニュアル　P88の「6,1　レコード」を参照）が用いられているわけですが，固定長レコードでは，各レコードにあまりの領域ができることが多いのです．例えば，1レコードが1人の人間の情報を記憶するのに使われ，1レコードのフィールドを，次のように決めたのでは，256バイト中116バイトも無駄にしていることになります．

名前	住所	電話	職業	あまり
20	50	20	50	116

まあ，どうしても，名前は20文字，住所は50文字，電話は20文字，職業50文字必要ならば話は別ですが，何とかならないものでしょうか．この例では，140バイト必要なのですが，どれかを少し減らして128バイトにすれば，1レコードに2人分入れられるようにできそうです．そこで次のように変更します．

1レコード

名前	住所	電話	職業	もう一人分
20	45	18	45	128

名前は短かくなると気分の悪いものですからそのまま，住所を5文字，電話を2文字，職業を5文字削ってみました．ここでは，例として決めただけで，実際にはよく調べてから決めないと後で困ります．というわけで，何とか128文字に詰め込むことができました．次の問題は，名前とレコード番号の対応です．1レコードに1人の情報を記憶させた場合は，レコード番号と人間は1対1に対応していましたが，1レコードに2人の情報を記憶させた場合は，1対1対応が成立しません．そこで考えられるのは，今までのレコード番号を2倍して，偶数は前半分，奇数は後半分と対応させる方法です．こうすれば，人間の番号とその人のデータが1対1に対応することになります．例えば，

```
100 '
110 ' divided record demo
120 '
130 OPEN "test" AS #1
140 FIELD #1, 20 AS NO$, 45 AS AO$, 18 AS TO$, 45 AS OO$,
              20 AS N1$, 45 AS A1$, 18 AS T1$, 45 AS O1$
150 '
160 PRINT
170 INPUT"read or write";AN$
180 IF AN$="end" THEN END
190 INPUT"number";NB
200 C=NB MOD 2
210 R=NB¥2+1
220 IF LEFT$(AN$,1)="r" THEN 320
230 '
240 GET #1,R
250 INPUT "name      ";N$
260 INPUT "address   ";A$
270 INPUT "telephone";T$
280 INPUT "work      ";O$
290 IF C THEN LSET N1$=N$:LSET A1$=A$:LSET T1$=T$:LSET O1$=O$
         ELSE LSET NO$=N$:LSET AO$=A$:LSET TO$=T$:LSET OO$=O$
300 PUT #1,R
310 GOTO 160
320 '
330 GET #1,R
340 IF C THEN N$=N1$: A$=A1$: T$=T1$: O$=O1$
         ELSE N$=NO$: A$=AO$: T$=TO$: O$=OO$
350 PRINT "name.............";N$
360 PRINT "address..........";A$
370 PRINT "telephone........";T$
380 PRINT "work.............";O$
390 GOTO 160
```

のようなプログラムになります．このプログラムは，マニュアル　97ページの Fig 26 を，1レコード2人に，変更したものです．140行の FIELD 文で，1レコードの区分けをしています．0の付いている文字変数は，人間の番号が偶数である場合，1の付いている文字変数は，人間

の番号が奇数である場合に使われます. 190行で人間の番号を入力します. 200行で入力された番号が, 偶数であるか奇数であるか判定するための変数Cを計算しています. 偶数であればCの値は0, 奇数であれば1が代入されます. 210行で対応するレコード番号Rを計算しています. ここで, 注意しなければならないのは, 番号の若い順からディスクに書き込んでいく場合はよいのですが, ランダムに書き込んでいく場合は, すでに記憶されているデータを壊してしまわないように, 合成する必要があります. それで, PUT# する前に GET# して, 前の情報と混ぜているわけです.

このように, 単純文字変数を使うと, if 文がしつこく現われてよくないので, 文字配列を使えばもっとスマートになります.

```
100 '
110 ' divided record demo 2
120 '
130 OPEN "test" AS #1
140 FIELD #1, 20 AS N$(0), 45 AS A$(0), 18 AS T$(0), 45 AS O$(0),
            20 AS N$(1), 45 AS A$(1), 18 AS T$(1), 45 AS O$(1)
150 '
160 PRINT
170 INPUT"read or write";AN$
180 IF AN$="end" THEN END
190 INPUT"number";NB
200 C=NB MOD 2
210 R=NB¥2+1
220 IF LEFT$(AN$,1)="r" THEN 320
230 '
240 GET #1,R
250 INPUT "name     ";N$
260 INPUT "address  ";A$
270 INPUT "telephone";T$
280 INPUT "work     ";O$
290 LSET N$(C)=N$:LSET A$(C)=A$:LSET T$(C)=T$:LSET O$(C)=O$
300 PUT #1,R
310 GOTO 160
320 '
330 GET #1,R
340 PRINT "name............";N$(C)
350 PRINT "address.........";A$(C)
360 PRINT "telephone.......";T$(C)
370 PRINT "work............";O$(C)
380 GOTO 160
```

FIELD 文の働きがよく理解できている方には, 次のプログラムが最もスマートかもしれません.

```
100 '
110 ' divided record demo 3
120 '
130 OPEN "test" AS #1
140 '
150 PRINT
160 INPUT"read or write";AN$
170 IF AN$="end" THEN END
180 INPUT"number";NB
190 C=NB MOD 2
200 R=NB¥2+1
210 FIELD #1, C*128 AS D$, 20 AS N$, 45 AS A$, 18 AS T$, 45 AS O$
220 IF LEFT$(AN$,1)="r" THEN 310
230 '
240 GET #1,R
250 INPUT "name     ";I$:LSET N$=I$
260 INPUT "address  ";I$:LSET A$=I$
270 INPUT "telephone";I$:LSET T$=I$
280 INPUT "work     ";I$:LSET O$=I$
290 PUT #1,R
300 GOTO 150
310 '
320 GET #1,R
330 PRINT "name..............";N$
340 PRINT "address...........";A$
350 PRINT "telephone.........";T$
360 PRINT "work..............";O$
370 GOTO 150
```

この例で, 文字変数 DS$ はダミーです, つまり間を空けるためにのみ使っているので, 他で使ってない文字変数ならなんでもかまいません.

この例にかぎらず, ランダムアクセスファイルを使いたいのだけれど, 1つの情報単位が256バイトよりも, あまりに小さすぎて, ディスクの使用効率がよくないと悩んでいる方は, 同様な方法で解決できます. 例えば, 情報単位が64バイトならば, 1レコードを4分割すればよいのです. CとRの算出は次のようにします.

C=NB　MOD 4
R=NB¥4＋1

3．2．6　レコードの連結

前とはまったく逆の場合を考えてみましょう. すなわち, ランダムアクセスファイルを使いたいのだけれど情報単位が256バイトよりも多い場合です.

最も単純なのは, すべての情報単位が256バイトより大きいのだけれど, 同じ大きさである場合です. 例えば, すべて500バイトづつ必要である場合には, それぞれ2レコードづつ割り振ればよいのです. 第1ブロックは1番レコードと2番レコード, 第2ブロックは3番レコードと4番レコード, というように割振るのです.

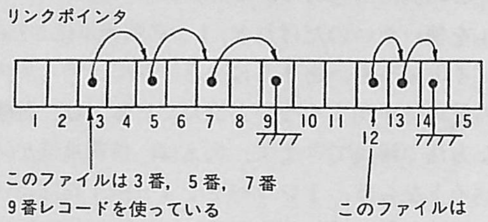
record＝number＊2－1

field#1,　100　as A$, 100 as　B$, 56　as C1$
　get
field#2,　44　as C2$,　100 as　D$,　100　as E$
　get

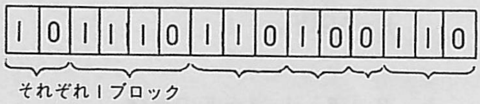
ところが, 1人当りに必要なレコード数が人によって違う場合です. これは, リスト構造に

よって解決できます．具体的には，一次元配列を用意してその中に次のレコード番号を入れておく，あるいは連続してレコードを占有している場合には，連がるのか連がらないのかの印だけでよいのです．

レコードの連結（配列を使う方法）

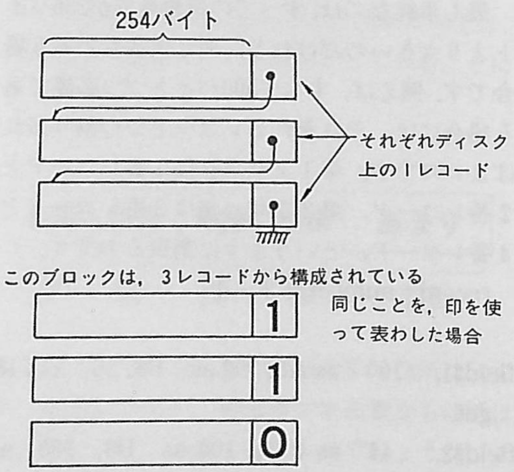


マーク法



ディスク上に，そのポインタを作っておくことも考えられます．

レコードの連結（ディスク上に書く方法）



3つのレコードを2人で使うとか，ディスクをせこく使おうとするときりがありません．あまり複雑にすると，ランダムアクセスファイルのメリットが薄れてしまいます．ほどほどにしておきましょう．

3．3　ファイルの属性-その意味と用法

ファイルには名前と拡張子があることを，前に説明しました．名前と拡張子はファイルを識別するだけのもので，Disk BASIC 自身がそのファイルについて特別な配慮はしません．ファイルの属性は，Disk BASIC に，そのファイルが特別なものであることを示すために用いられます．

ファイルの属性には，2つの種類があります．

(1) 書き込み禁止
(2) 書き込み確認

どちらの属性も重要なファイルに付けます．書き込み禁止属性は，完成したプログラムに付けて，不用意にプログラムが壊わされるのを防ぎます．Disk　BASIC は，あるファイルを作成するとき，すでに同一名のファイルが存在している場合には，そのファイルを自動的に消去し，同じ名前の新しいファイルを作ります．この機能はいちいち前のファイルを kill する必要がなくて便利ですが，うっかり同じ名前でプログラムを save して，大切な前のプログラムを壊わしてしまった，などということになりかねません．そのためにこの属性があるのです．書き込み確認属性は，プログラムにも付けますが，特にデータファイルに付けて用います．正しくデータが書けたかどうか，その場で確認するので，後でそのデータがうまく書かれていなかった，などというような事故を防ぎます．

これらの属性をファイルに付けるには，具体的にどのようにすればよいのでしょうか．それには SET コマンドを使います．

SET 〈ファイル名〉，〈属性文字〉

属性文字は，PまたはRで，次のように対応しています．

P…………書き込み禁止
R…………書き込み確認

上の2つの文字以外は，属性の解除に用いられます．ここで注意しなければならないのは，PとRは大文字でなければならないことです．

たとえば，〝format〞プログラムは，もう書き変えることのない完成したプログラムですから，書き込み禁止属性を付けてみましょう．

```
SET "format", "P"
```

このファイルを強引に殺そうとしますと

```
kill "format"
File write protected
```

と，エラーになり，殺せないことがわかります．

ここで注意したいのは，書き込み禁止属性は，Disk BASICによって，ソフト的に有効になっていますので，ドライブを直接動作させるコマンド，たとえば，DSKO$などには無力であることです．したがって，付属のシステムディスケットには，ハード的な書き込み禁止，すなわち，ライトプロテクトノッチにシールを張ることをおすすめします．ファイルの属性を解除してみましょう，

```
SET "format", "Q"
```

もう一度いまのプログラムを走らせてみれば，属性が解除されたことがわかるでしょう．

ドライブにも属性を与えることもできます．

SET 〈ドライブ番号〉，〈属性文字〉

以後作られるファイルは，その属性を持ちます．この「以後作られる」というところが大切で，つまり，前に作られたファイルに，その属性が付くわけではありません．ドライブに属性が付くのであってディスクに属性が付くわけではないのです．以後removeするまでその属性は有効です．ディスクの属性は，IDセクタの最初の1バイトの値で設定され，mountするときに，その値が読み出され，そのディスクの入っているドライブの属性が指定されるわけです．したがって，DSKO$コマンドによって，IDフィールドの最初のバイトに10（16進）を書き込んでおけば，そのディスクに書き込むことはできなくなります．たとえば，

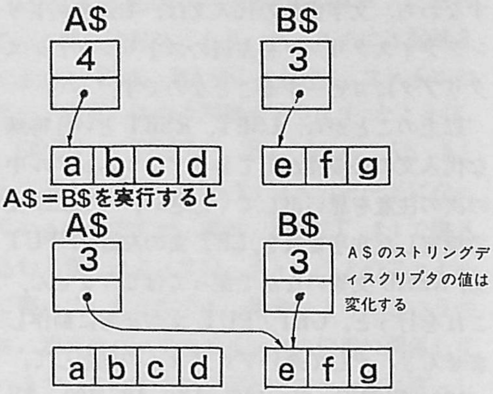
```
FIELD #0, 1 AS AT$
D$=DSKI$ (1, 18, 13)
LSET AT$=CHR$ (&H10)
DSKO$ (1, 18, 13)
```

とすれば，第1ドライブに入っている，ディスケットには書込み禁止が付くわけです．

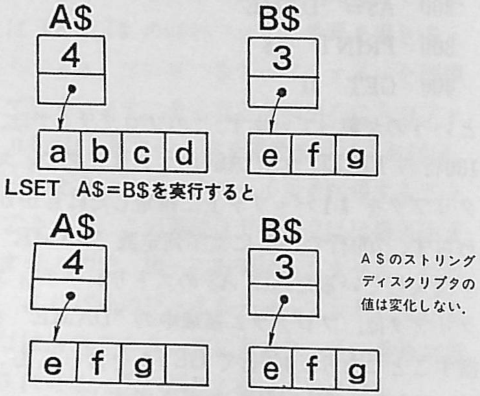
このディスクに書きたい場合には，SET文で，そのディスクの入っているドライブの書き込み禁止属性を解除してやればよいのです．しかし，ディスクの属性が変わるわけではありません，再びmountしたときには，書き込み禁止属性が指定されるのです．このように，書き込みに，ワンステップおくことによって，不用意にファイルを破壊する事故を防ぐことができます．プログラムディスクとデータディスクを分けておいて，プログラムディスクには，書き込み禁止属性を付けておくのもよいでしょう．

その他に，ファイル番号に属性を付けることができます．

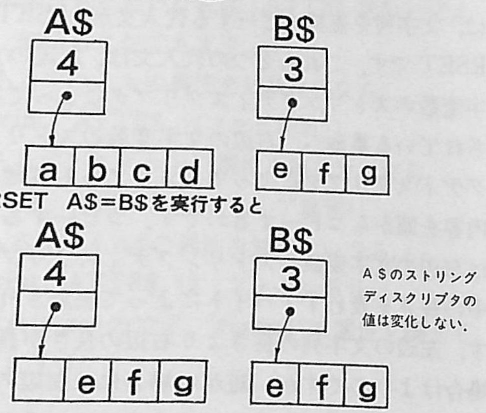
SET# 〈ファイル番号〉，〈属性文字〉



LETとLSET・RSETの違い



LETとLSET・RSETの違い



3．5　シーケンシャルファイルのランダムアクセス

シーケンシャルファイル中の特定のデータのみを引き出すには，頭から順序よく読んでいって，必要なデータが出て来るまで，不要なデータを捨てながら，待たなくてはなりません．この方法は，ファイルが短かければよいのですが，長いファイルの後の方のデータを引き出すのはたいへんな時間がかかります．そこを何とか改善できないものでしょうか．それは，GET#文をシーケンシャルファイルに用いることによって可能になるのです．もちろん，そのデータの入っているレコード番号が解からなくてはなりません．その値を教えてくれるのが，LOC関数です．LOC関数も，ランダムファイルのための関数ですが，シーケンシャルファイルに用いると，それまで読み書きしたセクタの総数，すなわち，データが入るレコード番号より1小さい値を与えます．このLOC関数を使って，ランダムファイルで説明したような，対応表を作ればよいのです．新しくシーケンシャルファイルを作る場合には，対応表を作りながらファイルを作ればよいのですが，すでに，ディスク上にあるファイルから対応表を作るには，一度読まなければなりません．一度，対応表を作ってしまえば，後はランダムアクセス並に，アクセスできるのです．といっても，GET#文でたしかに，必要なデータは，バッファの中に取り込まれましたが，ランダムファイルのように固定のフィールドが定まってないので，どこからが必要なデータか解かりません．ですから，少しは，シーケンシャルに読まなければなりませんが，256バイトの中で見つけるのですから，ファイルを頭から読んでいくのとは，大ちがいです．1レコードの中に複数のデータブロックが入っているときには，番号とか名前とか識別子をデータブロック中に，入れておく必要があります．たとえば，シーケンシャルファイルを作るとき

に，
……名前，住所／名前，住所／………
のように入れておけば，一つ読むたびに比較して，求める住所を引き出せます．

以上のような話は，発想の転換で，シーケンシャルファイルのコマンドをランダムファイルに使い，ランダムファイルのコマンドをシーケンシャルファイルに使ってもよいではないか，ということです．このようなことは，ある程度 Disk　BASIC を理解してからでないと，思わぬ事故を起こします．くれぐれも注意して使いながら，もっと他の使用法も考えてみてください．

4. 応用の実際

4.1　モジュール化プログラミング

4.1.1　モジュール化プログラミングとは

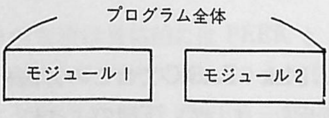
プログラムが長くなって，どうしてもメモリに収まらないことがあります．こんな時の対応策として考えられる方法には

＊REM 文をできるだけ省略する．

＊配列はできるだけ少く，また倍精度よりは単精度を，できれば整数型の配列にする．

等がありますが，ディスクが使える場合には，プログラムをモジュール化するテクニックが使えます．

プログラムのモジュール化とは，1本のプログラムを分割して何本かのプログラムにして実行する方法です．



図のように，プログラム全体がメモリ内に収まらない時，1本のプログラムを2本以上の部分（モジュール）に分割して，モジュール別にメモリ内にロードして実行するわけです．

上の例では，1本のプログラムを2個のモジュールに分割しています．このプログラムを実行する時は，最初のモジュールをメモリにロードして実行し，モジュール1の動作が終了した時点で，ディスクからモジュル2がロードされて実行がモジュール2に移るわけです．

例ではプログラムを2個に分割しましたが，もっと多数に分割できますから，プログラムの長さの制限はディスクの容量の制限ということになります．

この方法は，ディスクの容量と速度ならではの方法といえるわけです．

4.1.2　実現する方法

さて，実際にこの方法をどのように実現する

か考えてみましょう．

まず最初の問題は，どのようにしてモジュール1の終了から，モジュール2の実行に入るかです．モジュール1がENDで終了するとBASICはコマンド待ちの状態になりますからEND文は使えそうもありません．

何か，実行を他のプログラムに移すステートメントがないか，2章を開いて探してみましょう．あるでしょう．RUN です．N-BASIC はコマンドもプログラム中に書くことができますから，モジュール1の最後を

RUN 〝モジュール2″

で終えれば，モジュール1の実行後に即座にモジュール2が動きだすでしょう．この調子で，プログラムを実行していけば，エラーになるか，END文を実行するまでコマンド待にはならないはずです．

4．1．3　モジュール化プログラムの問題点

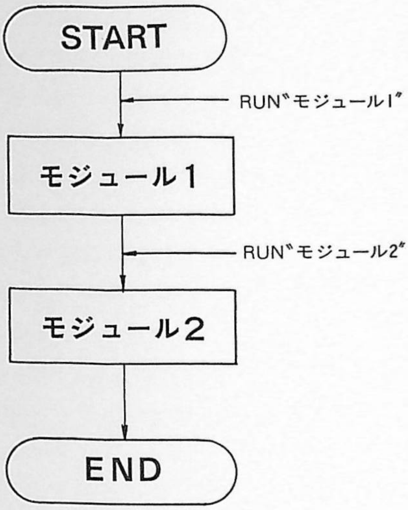
プログラムをモジュール別に書くことができることは分りました．が，少々問題点があるようです．もう1度第2章のRUNの説明を読んで下さい．プログラムや変数がクリアされると説明されています．プログラムがクリアされるのはかまいませんが，プログラムの実行の途中で全ての変数がクリアされたのではプログラムが成立しません．何らかの手段を用いて変数の引き渡しをしなければならないようです．

もう1つ，問題点とは言えないまでも不満な点があります．

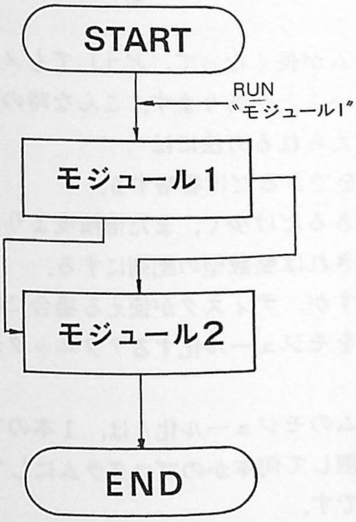
RUN 〝モジュール2″

を実行すると，モジュール2がロードされて動き出しますが，動き出すのはモジュール2の最初の行からだということです．

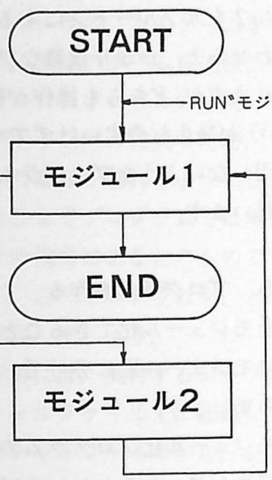
この実行の手段は次の図のようなものです．



ところが，実際のプログラムングでは時として モジュール2の入口を2つ以上作りたくなります．下図のようにです．



残念ながら，Disk　BASICではこの方法を許してはくれません．まして，下図のように，



モジュール2を，モジュール1のサブルチーンのように使用することも許してはくれないのです．

4．1．4　問題点をのりこえて

まず変数をモジュール2に引き渡す方法です．2章の RUN の説明を読むと，RUN を実行するとプログラム領域と変数領域をクリアすると書いてありますから，クリアされない領域すなわちプログラム領域と変数領域以外の場所に変数の値を保存すればよいわけです．

そこで考えられる方法をあげてみます．これは2章にも書きましたが，

1)プログラム領域と変数領域を縮めて，余ったスペースを使用する．

2)変数引き渡し用のファイルを作る．

3)ファイル用に確保されるバッファ領域を使用する．

1)の方法は具体的には PEEK と POKE を使用するわけですが，欠点があります．PEEK とPOKE は1回に1バイトの値（0－255の整数）しか扱えません．従って単精度実数や倍精度実数，さらに文字変数を保存するためにはプログラムが複雑化してきます．

2)この方法は最も ノーマルの方法といえるでしょう．1つのモジュールが終了する際に，必要な変数の値をファイルに保存して，次のモジュールでそのファイルから値をもどすわけです．

具体的にはランダムファイルと，シーケンシャールファイルを使用することになります．特に，ランダムファイルの場合には，ファイルを閉じる必要がありませんから，RUN"モジュール2"，R　の，Rオプションのついた RUN が使用できます．

3)この方法は，2)の方法を簡略化した方法です．ランダムファイルを使用して値を引き渡す場合値の量が256バイトに満たない時に，ディスクに書き込む動作を省略して，バッファ領域に書き込むだけで値を渡す方法です．2)の方法の場合，ディスクへの書込動作が行われますが，この方法ではディスクへは書込み動作は行われません．従って速度は向上するはずです．

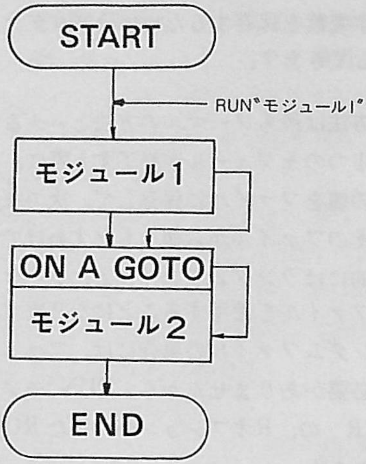
変数の引き渡し方法がおよそ理解できたら次はモジュール間のプログラムの流れについて考えてみましょう．モジュール2の入口を2つにすることにせよ，モジュール2をサブルーチンと扱うことにせよ，Disk　BASIC が許していないことをするわけですから，若干見にくい構成になります．

モジュール2の入口を2つにする方法は，モジュール1からモジュール2への入口を示す値を引き渡すことで解決できるでしょう．

例えば，モジュール2の先頭を

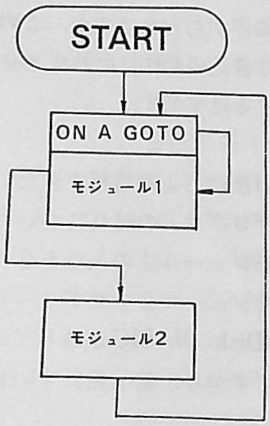
ON　A　GOTO●，●，●，………

にしておけば，モジュール1から変数Aの値を渡すことで解決できると思います．



この操作は上図のように書けるでしょう．

同様な考え方で，モジュール2をサブルーチンとして扱うことも可能になります．下図を見て下さい．



モジュール2をサブルチーンとして呼ぶことには問題がないのですが，モジュール1へ戻る時に問題になります．そこでモジュール2へ実行を移す時にAの値をモジュール2へ渡し，その値をそのままモジュール1へ戻る時に返してもらって，モジュール1へ入る場所を知るわけです．

モジュール2の入口を2つにするテクニックと，モジュール2をサブルーチンにするテクニックを組み合わせると，かなり複雑なプログラムの構造を作れますが，どちらも操作が複雑で，バグ（まちがい）が発生しやすいはずですから，できるだけ使用しないようなプログラムを書いた方がよいと思います．

4．1．5　プログラムを作る

プログラムをモジュール化できることが分ったので，実際にモジュール別に分けて，プログラムを作ってみましょう．

本書では，モジュール化プログラムの応用例として，ジョブスケジューラとシェルピンスキー曲線を描くプログラムを作ってみました．本来なら，プログラムをモジュール化することは，あまり良い結果を招きませんから，メモリに収まるプログラムを進んでモジュール化することは避けるべきですが，モジュール化プログラムの応用例のプログラムがモジュール化されていなくては話になりません．ここで示したプログラムは，読者の方が，しかたなくプログラムをモジュール化する際の参考例と考えて下さい．

4．1．6　例#1　ジョブ・スケジューラ

一般的に言われることですが，マイコンを始めた読者の最大の悲劇は，マイコンに夢中になって勉強しなくなることです．このことは学生のみならず社会人についても言えるでしょう．ゲームに夢中で徹夜してしまうこともありがちです．

そこで，このジュブスケジューラは，そんなあなたに注意を促す意味で作ったプログラムです．

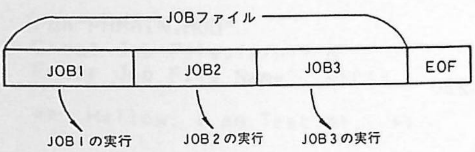
このプログラムは短かいのですが，おもしろい機能をもっています．最初にPCに何の仕事をするか教えておき，それらの仕事を連続して行なわせるわけです．ですから，勉強や仕事の

合間にちょっとゲームで遊ぶような時に，このスケジューラで，どのゲームをするか指定しておけば，指定のゲームを連続して行い，終了するまでコマンドレベルには戻らないわけです．

コマンドレベルに戻った時，できれば自動的に PC の電源が切れるようにしたかったのですが，そこまでプログラムでは実行できませんから，その役目はあなたのものです．

さて，このプログラムを，どのように実現したか説明しましょう．

基本的には，実行するジョブ名を1本のシーケンシャルファイル上に記録しておき，このシーケンシャルファイルから，ジョブ名を読んで実行することにしました．



こうすると，ジョブ名を読んでスタートさせるメイルルーチンが必要です．メインルーチンでは，次の動作をします．

＊ジョブファイルが終りなら，コマンドレベルに戻る．

＊ジョブ名を読んで，そのファイルがディスケット中に存在するなら，そのファイルを読んで実行する．

＊ジョブ名で指定されたファイルが存在しないなら，その旨を表示して，次のジョブ名を読む．

次に，ジョブファイルを作るプリメインルーチンについて説明しましょう．

このルーチンは実行するファイル名をシーケンシャルファイルに積むだけですから簡単です．ただ，一度作ったジョブファイルを何度でも実行できるように，壊さないことにしました．

最後に，ジョブスケジューラの下で実行するファイルについて説明します．プログラムをジョブスケジューラの下で実行させるのには2つの条件があります．

＊プログラムの最後は，END 文ではなく，メインルーチンを起動する形でなければなりません．

＊プログラムの実行中も，ジョブファイルは開いたままですから，ジョブファイルのファイル番号は，このプログラム中では使えません．

このプログラムは，最後をメインルーチンを呼ぶ形になりますから，END 文で終るプログラムと区別する意味で，ファイルネームに拡張子をつけ，他と区別することが望ましいでしょう．

実際の使い方を説明します．PRMAIN プログラムを動かすと最初に，

Create Job File (y／n) ?

とたずねてきます．存在するジョブファイルを実行する時にはnを，これからジョブファイルを作る時には y を入力して下さい．

次に PC は

Enter Job File Name ?

と聞いてきます．ここでジョブファイル名を入力します．この後，最初の質問に n で答えた場合，この時入力したジョブファイル内のジョブを実行しはじめます．

最初の質問に y で答えた場合には，この後

JOB#1＝

と出力してきますから，実行する順にプログラムファイル名を入力して下さい．1個入力する毎に

JOB#2＝………

JOB#3=………
と聞いてきます.

最後の JOB を入力した後は, end と入力すれば, 最初の JOB から実行を開始します.

```
100 '
110 ' Job Schedular
120 '             PRMAIN Program
130 '
140  INPUT "Creat Job File.(y/n)";A$:A$=LEFT$(A$,1)
150 '
160 IF A$="y" THEN GOTO 200
      ELSE IF A$="n" THEN GOTO 290
              ELSE GOTO 140
170 '
180 ' Make Job File
190 '
200    INPUT "Enter Job File Name";JOB$
210    OPEN JOB$ FOR OUTPUT AS #1:NUM=1
220 '
230      PRINT "Job #";NUM;
240      LINEINPUT"=";A$
250  IF A$="end" THEN CLOSE #1:GOTO 330
       ELSE PRINT #1,A$:NUM=NUM+1:GOTO  230
260 '
270 ' Job File are Already Exist
280 '
290 INPUT "Enter Job File Name";JOB$
300 '
310 '  Goto Main :Exicute Job
320 '
330 OPEN JOB$ FOR INPUT AS #1
340 RUN "MAIN.MOD",R
```

```
100 '
110 '  Job Schedular
120 '             MAIN Program
130 '
140    ON ERROR GOTO 230
150    WIDTH 80,25
160 '
170 ' Exic next Job
180 '
190     IF EOF(1) THEN CLOSE#1:PRINT "*** JOB END ***":END
                ELSE INPUT #1,A$:RUN A$,R
200 '
210 '  File not foind. Print Message
220 '
230     IF ERR=53 THEN PRINT"job **";A$;"**  is not found.":RESUME 190
          ELSE IF ERR=62 THEN RESUM 150
                  ELSE ON ERR GOTO 0
```

ジョブスケジューラ実行例

```
100 '
110 ' Test Program #1 for Job Schedular
120 '
130         PRINT
140         PRINT "**  Hellow! I am Test #1.  **"
150         PRINT
160 '
170 ' Test #1 End
180 '
190         RUN "MAIN.MOD",R
```

```
100 '
110 ' Test Program #2 for Job Schedular
120 '
130         PRINT
140         PRINT "**  I am Test #2.  Good Bye  **"
150         PRINT
160 '
170 ' Test #2 End
180 '
190         RUN "MAIN.MOD",R
```

```
run"PRMAIN.MOD"
Creat Job File.(y/n)? n
Enter Job File Name?  ASCII
 **  Hellow! I am Test #1.  **

 **  I am Test #2.  Good Bye  **

 *** JOB END ***
Ok
```

4．1．7例 ＃2シェルピンスキー曲線

まずシェルピンスキー曲線について説明しましょう．

N次のシェルピンスキー曲線を描く手続きは

A (N) ↘B (N) ↙C (N) ↖D (N) ↗

と定義されます．↗は右上りの単位長の直線を描くことを示しています．又，A (N) ~D (N) は次のように定義されています．

1) N＝0の時　何もしない

2) N＞ 0の時

A (N) : A (N-1) ↘B (N-1) →→D (N-1) ↗A (N-1)

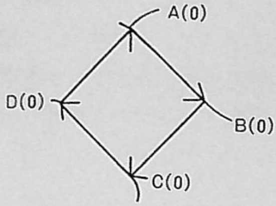
B (N) : B (N-1) ↙C (N-1) ↓↓A (N-1) ↘B (N-1)

C (N) : C (N-1) ↖D (N-1) ←←B (N-1) ↙C (N-1)

D (N) : D (N-1) ↗A (N-1) ↑↑C (N-1) ↖D (N-1)

御覧の様にA (N) の定義にA (N-1) が使用されるという，再帰的に定義された曲線なわけです．

頭の中だけで考えると，複雑でわからなくなりますが，試みに0次のシェルピンスキー曲線を考えると，



となることは分るでしょう．あとはプログラムを走らせてみて下さい．

さて，この定義をプログラミングする方法について考えてみましょう．まず最初に頭に浮ぶのは，A (N)，B (N)，C (N)，D (N) をサブルーチンにしてしまうことです．更に単位長の直線を引く部分もサブルーチンにすればよいでしょう．A (N) ~D (N) をサブルーチンにすると，このサブルーチンは再帰呼出しすることになりますから，ネストの深さが心配になります．ネストはN次のシェルピンスキー曲線でN＋1回になることは明白なので，5～6次のシェルピンスキー曲線までならモジュール化する必要はないと思いますが，モジュール化プログラムの実例を示す意味で，今回は強引に，各サブルーチンをモジュール別に分離したプログラムを作ってみました．

プログラムは次の6つのモジュールから構成されています．

＊イニシャライズプログラム
＊メインプログラム
＊A (N)プログラム
＊B (N)プログラム
＊C (N)プログラム
＊D (N)プログラム

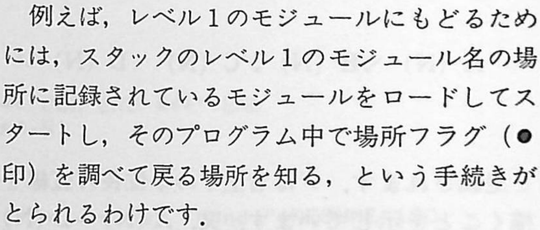
単位長のラインを引くプログラムは独立せずに各プログラム中に含ませました．
このプログラム中で使用する変数は，ファイル番号#1のランダムバッファを使用しています．

その変数は下の通りです．

DEP：現在のスタックの深さを示す．
N　：何次のシェルピンスキーを描いているかを示す．(N＋1の価になっている)
H　：単位長の直線の長さを示す．
X　：現在のプロット位置 (X軸)
Y　：現在のプロット位置 (Y軸)

これらの変数はイニシャライズプログラムで初期値がセットされるわけです．

次にスタックについて説明します．
このプログラムは，イニシャライズを除く各ルーチンが4回のサブルーチンコールを行い，互いに呼びあう形になるので，サブルーチンからもどる時には，どのモジュールのどこへもどるのか，2つの情報をスタックに積まなければなりません．そのため，スタックもランダムバッファ上に作りました．
スタックの構造は下のようになっています．



例えば，レベル1のモジュールにもどるためには，スタックのレベル1のモジュール名の場所に記録されているモジュールをロードしてスタートし，そのプログラム中で場所フラグ（●印）を調べて戻る場所を知る，という手続きがとられるわけです．

このプログラムを実際に動かす時には，必ずイニシャライズプログラムからスタートして下さい．他から動作させても正常な動作はしません．

イニシャライズをスタートすると，

S（0，1，2，3）?

と聞いてきます．これは何次のシェルピンスキーを描くか聞いているわけです．0～3の好きな数を押してリターンして下さい．ディスクがカッコンカッコン始めます．即座に描き始めないからといって，あせらず見守ると描きだします．これも1辺ずつ，ゆっくりとです．

このプログラムは，モジュールのロードだけで，0次の時に10回，1次で42回，2次で172回，3次では682回ディスクをアクセスします．遅いのは当然なのです．実は4次のシェルピンスキーまでなら，PCのグラフィックでも実現は可能と思われるのですが，速度の面を考えて断念したわけです．3次を描かせるのにだいたい40分必要だったのです．4次では2728回ディスクをアクセスしますから，4～5倍の時間が必要で，現実的ではありません．先にも書いたように，このプログラムはモジュール化しなくても動かせますから，自分で作って描かせてみて下さい．

シェルピンスキー曲線のイニシャライズ
プログラム

```
100 '
110 '
120 ' This is Initrize Program of SIERPINSKY
130 '
140  WIDTH 80,25
150 '
160 ' Initrize Variable Space
170 '
180  OPEN "STACK" AS #1
190    FIELD #1,2 AS DEP$,2 AS N$,2 AS H$,2 AS X$,2 AS Y$
200    FOR I=0 TO 9
210        FIELD #1,I*11+10 AS DUMMY$(I),9 AS PRG$(I),2 AS FLG$(I)
220    NEXT I
230 '
240 ' Serect N
250 '
260   INPUT "   S(0,1,2,3) " ;I:I=INT(I)
270      IF (I<0) OR (I>3) THEN GOTO 260
280   H=24¥(2^I)
290 '
300 ' Initrize Variable
310 '
320     LSET DEP$=MKI$(0)
330     LSET H$=MKI$(H)
340     LSET N$=MKI$(I+1)
350     LSET X$=MKI$(25+H)
360     LSET Y$=MKI$(0)
370     LSET FLG$(0)=MKI$(0)
380 '
390 ' Initrize Display
400 '
410   PRINT CHR$(12)
420   CONSOLE 0,25,0,0
430   COLOR   0,0,1
440 '
450 ' GOTO MAIN PROGRAM
460 '
470  RUN "SIERP",R
```

```
110 '
120 ' This is Main Program of SIERPINSKY
130 '
140 '
150    FIELD #1,2 AS DEP$,2 AS N$,2 AS H$,2 AS X$,2 AS Y$
160     FOR I=0 TO 9
170         FIELD #1,I*11+10 AS DUMMY$(I),9 AS PRG$(I),2 AS FLG$(I)
180     NEXT I
190 '
200    X1=CVI(X$):Y1=CVI(Y$):I=CVI(DEP$):J=CVI(N$):H=CVI(H$)
210 '
220 'ON FLG BRANCH
230 '   FLG=0 : first entry
240 '      =1 : return from A(n)
250 '      =2 : return from B(n)
260 '      =3 : return from C(n)
270 '      =4 : return from D(n)
280 '
290     ON CVI(FLG$(I)) GOTO330,360,390,420
300 '
310 '
320        LSET FLG$(I)=MKI$(1):CALL$="SIERPA":GOTO 550 ' Gosub A(n)
330           X2=X1+H:Y2=Y1+H:GOSUB 500                ' line
340 '
350        LSET FLG$(I)=MKI$(2):CALL$="SIERPB":GOTO 550 ' Gosub B(n)
360           X2=X1-H:Y2=Y1+H:GOSUB 500                ' line
370 '
380        LSET FLG$(I)=MKI$(3):CALL$="SIERPC":GOTO 550 ' Gosub C(n)
390           X2=X1-H:Y2=Y1-H:GOSUB 500                ' line
400 '
410        LSET FLG$(I)=MKI$(4):CALL$="SIERPD":GOTO 550 ' Gosub D(n)
420           X2=X1+H:Y2=Y1-H:GOSUB 500                ' line
430 '
440 'All End
450 '
460     CLOSE #1:KILL "STACK": LOCATE 0,23,1: END
470 '
480 'Line (X1,Y1)-(X2,Y2) and  X1:=X2  Y1:=Y2
490 '
500     LINE (X1,Y1)-(X2,Y2),PSET,0
510     LSET X$=MKI$(X2):LSET Y$=MKI$(Y2):RETURN
520 '
530 'GO TO SUBROUTINE
540 '
550     LSET DEP$=MKI$(I+1):LSET FLG$(I+1)=MKI$(0)
560     LSET PRG$(I)="SIERP":RUN CALL$,R
```

メインプログラム

A (n) ↘　B (n) ↙　C (n) ↖　D (n) ↗

で構成されています.

```
100 '
110 '
120 ' This is Subroutine A(n) of SIERPINSKY
130 '
140 '
150     FIELD #1,2 AS DEP$,2 AS N$,2 AS H$,2 AS X$,2 AS Y$
160      FOR I=0 TO 9
170          FIELD #1,I*11+10 AS DUMMY$(I),9 AS PRG$(I),2 AS FLG$(I)
180      NEXT I
190 '
200     X1=CVI(X$):Y1=CVI(Y$):I=CVI(DEP$):J=CVI(N$):H=CVI(H$)
210 '
220 'If J=I Then return
230 '
240     IF J=I THEN GOTO 480
250 '
260 'ON FLG BRANCH
270 '    FLG=0 : first entry
280 '       =1 : return from A(n-1)
290 '       =2 : return from B(n-1)
300 '       =3 : return from D(n-1)
310 '       =4 : return from A(n-1)
320 '
330      ON CVI(FLG$(I)) GOTO360,390,420,480
340 '
350         LSET FLG$(I)=MKI$(1):CALL$="SIERPA":GOTO 570 ' Gosub A(n-1)
360            X2=X1+H:Y2=Y1+H:GOSUB 520                 ' line
370 '
380         LSET FLG$(I)=MKI$(2):CALL$="SIERPB":GOTO 570 ' Gosub B(n-1)
390            X2=X1+2*H:Y2=Y1:GOSUB 520                 ' line
400 '
410         LSET FLG$(I)=MKI$(3):CALL$="SIERPD":GOTO 570 ' Gosub D(n-1)
420            X2=X1+H:Y2=Y1-H:GOSUB 520                 ' line
430 '
440         LSET FLG$(I)=MKI$(4):CALL$="SIERPA":GOTO 570 ' Gosub A(n-1)
450 '
460 'All End: Return
470 '
480     LSET DEP$=MKI$(I-1):RUN PRG$(I-1),R
490 '
500 'Line (X1,Y1)-(X2,Y2) and  X1:=X2  Y1:=Y2
510 '
520     LINE (X1,Y1)-(X2,Y2),PSET,0
530     LSET X$=MKI$(X2):LSET Y$=MKI$(Y2):RETURN
540 '
550 'GO TO SUBROUTINE
560 '
570     LSET DEP$=MKI$(I+1):LSET FLG$(I+1)=MKI$(0)
580     LSET PRG$(I)="SIERPA":RUN CALL$,R
```

A (n) モジュール
A(n−1)↘B(n−1)→→D(n−1)↗A(n−1)
で構成されています.

```
100 '
110 '
120 ' This is Subroutine B(n) of SIERPINSKY
130 '
140 '
150    FIELD #1,2 AS DEP$,2 AS N$,2 AS H$,2 AS X$,2 AS Y$
160     FOR I=0 TO 9
170        FIELD #1,I*11+10 AS DUMMY$(I),9 AS PRG$(I),2 AS FLG$(I)
180     NEXT I
190 '
200    X1=CVI(X$):Y1=CVI(Y$):I=CVI(DEP$):J=CVI(N$):H=CVI(H$)
210 '
220 'If J=I Then return
230 '
240    IF J=I THEN GOTO 480
250 '
260 'ON FLG BRANCH
270 '   FLG=0 : first entry
280 '      =1 : return from B(n-1)
290 '      =2 : return from C(n-1)
300 '      =3 : return from A(n-1)
310 '      =4 : return from B(n-1)
320 '
330    ON CVI(FLG$(I)) GOTO360,390,420,480
340 '
350       LSET FLG$(I)=MKI$(1):CALL$="SIERPB":GOTO 570 ' Gosub B(n-1)
360         X2=X1-H:Y2=Y1+H:GOSUB 520                   ' line
370 '
380       LSET FLG$(I)=MKI$(2):CALL$="SIERPC":GOTO 570 ' Gosub C(n-1)
390         X2=X1:Y2=Y1+2*H:GOSUB 520                   ' line
400 '
410       LSET FLG$(I)=MKI$(3):CALL$="SIERPA":GOTO 570 ' Gosub A(n-1)
420         X2=X1+H:Y2=Y1+H:GOSUB 520                   ' line
430 '
440       LSET FLG$(I)=MKI$(4):CALL$="SIERPB":GOTO 570 ' Gosub B(n-1)
450 '
460 'All End: Return
470 '
480    LSET DEP$=MKI$(I-1):RUN PRG$(I-1),R
490 '
500 'Line (X1,Y1)-(X2,Y2) and  X1:=X2  Y1:=Y2
510 '
520    LINE (X1,Y1)-(X2,Y2),PSET,0
530    LSET X$=MKI$(X2):LSET Y$=MKI$(Y2):RETURN
540 '
550 'GO TO SUBROUTINE
560 '
570    LSET DEP$=MKI$(I+1):LSET FLG$(I+1)=MKI$(0)
580    LSET PRG$(I)="SIERPB":RUN CALL$,R
```

B (n) モジュール

B (n−1) ↙C (n−1) ↓↓A (n−1) ↘

B (n−1)

で構成されています.

```
100 '
110 '
120 ' This is Subroutine C(n) of SIERPINSKY
130 '
140 '
150     FIELD #1,2 AS DEP$,2 AS N$,2 AS H$,2 AS X$,2 AS Y$
160      FOR I=0 TO 9
170           FIELD #1,I*11+10 AS DUMMY$(I),9 AS PRG$(I),2 AS FLG$(I)
180      NEXT I
190 '
200     X1=CVI(X$):Y1=CVI(Y$):I=CVI(DEP$):J=CVI(N$):H=CVI(H$)
210 '
220 'If J=I Then return
230 '
240     IF J=I THEN GOTO 480
250 '
260 'ON FLG BRANCH
270 '    FLG=0 : first entry
280 '        =1 : return from C(n-1)
290 '        =2 : return from D(n-1)
300 '        =3 : return from B(n-1)
310 '        =4 : return from C(n-1)
320 '
330      ON CVI(FLG$(I)) GOTO360,390,420,480
340 '
350         LSET FLG$(I)=MKI$(1):CALL$="SIERPC":GOTO 570 ' Gosub C(n-1)
360              X2=X1-H:Y2=Y1-H:GOSUB 520                ' line
370 '
380         LSET FLG$(I)=MKI$(2):CALL$="SIERPD":GOTO 570 ' Gosub D(n-1)
390              X2=X1-2*H:Y2=Y1:GOSUB 520                ' line
400 '
410         LSET FLG$(I)=MKI$(3):CALL$="SIERPB":GOTO 570 ' Gosub B(n-1)
420              X2=X1-H:Y2=Y1+H:GOSUB 520                ' line
430 '
440         LSET FLG$(I)=MKI$(4):CALL$="SIERPC":GOTO 570 ' Gosub C(n-1)
450 '
460 'All End: Return
470 '
480     LSET DEP$=MKI$(I-1):RUN PRG$(I-1),R
490 '
500 'Line (X1,Y1)-(X2,Y2) and  X1:=X2  Y1:=Y2
510 '
520     LINE (X1,Y1)-(X2,Y2),PSET,0
530     LSET X$=MKI$(X2):LSET Y$=MKI$(Y2):RETURN
540 '
550 'GO TO SUBROUTINE
560 '
570     LSET DEP$=MKI$(I+1):LSET FLG$(I+1)=MKI$(0)
580     LSET PRG$(I)="SIERPC":RUN CALL$,R
```

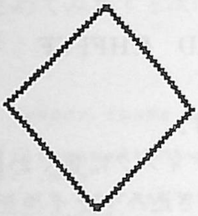
C (n) モジュール
C (n－1) ↖　D (n－1) ←←B (n－1)
↙C (n－1)
で構成されています.

```
100 '
110 '
120 ' This is Subroutine D(n) of SIERPINSKY
130 '
140 '
150    FIELD #1,2 AS DEP$,2 AS N$,2 AS H$,2 AS X$,2 AS Y$
160     FOR I=0 TO 9
170        FIELD #1,I*11+10 AS DUMMY$(I),9 AS PRG$(I),2 AS FLG$(I)
180     NEXT I
190 '
200    X1=CVI(X$):Y1=CVI(Y$):I=CVI(DEP$):J=CVI(N$):H=CVI(H$)
210 '
220 'If J=I Then return
230 '
240    IF J=I THEN GOTO 480
250 '
260 'ON FLG BRANCH
270 '   FLG=0 : first entry
280 '      =1 : return from D(n-1)
290 '      =2 : return from A(n-1)
300 '      =3 : return from C(n-1)
310 '      =4 : return from D(n-1)
320 '
330     ON CVI(FLG$(I)) GOTO360,390,420,480
340 '
350       LSET FLG$(I)=MKI$(1):CALL$="SIERPD":GOTO 570 ' Gosub D(n-1)
360          X2=X1+H:Y2=Y1-H:GOSUB 520               ' line
370 '
380       LSET FLG$(I)=MKI$(2):CALL$="SIERPA":GOTO 570 ' Gosub A(n-1)
390          X2=X1:Y2=Y1-2*H:GOSUB 520               ' line
400 '
410       LSET FLG$(I)=MKI$(3):CALL$="SIERPC":GOTO 570 ' Gosub C(n-1)
420          X2=X1-H:Y2=Y1-H:GOSUB 520               ' line
430 '
440       LSET FLG$(I)=MKI$(4):CALL$="SIERPD":GOTO 570 ' Gosub D(n-1)
450 '
460 'All End: Return
470 '
480     LSET DEP$=MKI$(I-1):RUN PRG$(I-1),R
490 '
500 'Line (X1,Y1)-(X2,Y2) and  X1:=X2  Y1:=Y2
510 '
520     LINE (X1,Y1)-(X2,Y2),PSET,0
530     LSET X$=MKI$(X2):LSET Y$=MKI$(Y2):RETURN
540 '
550 'GO TO SUBROUTINE
560 '
570     LSET DEP$=MKI$(I+1):LSET FLG$(I+1)=MKI$(0)
580     LSET PRG$(I)="SIERPD":RUN CALL$,R
```

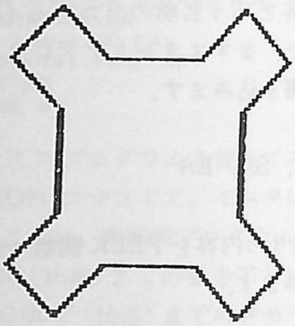
D (n) モジュール
D (n－1) ↗A (n－1) ↑↑C (n－1) ↖
D (n－1)
で構成されています.

（**注意**：このモジュール化プログラムは，1枚のディスケット中に納めて下さい．ディスケットはかならず，ドライブ1に入れて下さい.）

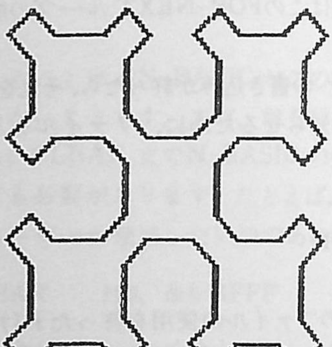
0次



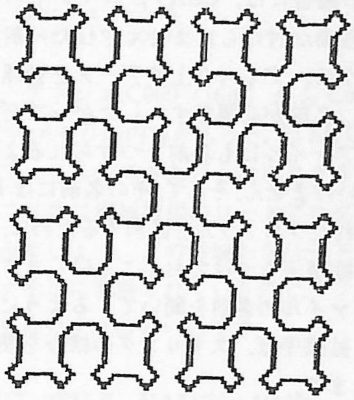
1次



2次



3次



4．2　機械語ファイルの作り方

PC-8001のモニタには，Wコマンド（ライトテープ）と，Lコマンド（ロードテープ）という2つの命令があって，メモリの内容をそのままの形でテープ（オーディオカセット）に保存しておくことができます．しかし残念ながら，ディスクに対してはメモリの内容を直接保存しておくコマンドが用意されていません．

それでも，N-BASICには，PEEK，POKEコマンドがあるので，簡単にそのためのプログラムを作ることができます．それではまず，メモリの内容をディスクに書き込むSAVEプログラムを作ってみましょう．

参考のために W コマンドが次のような形をしていたことを思い出しましょう．

```
*WXXXX, YYYY
    XXXX＝スタートアドレス
    YYYY＝エンドアドレス
```

つまり，W コマンドを使うときには，メモリの最初のアドレスと最後のアドレスを入力しなけ

ればならなかったのです．けれども，このWコマンドの場合には，CSAVEコマンドの時とは違って名前が付けられませんでした．前にも書いたように，ディスク上でデータを管理するためには，名前が必要です．したがって必然的に機械語ファイルにも名前をつけられるようにしないといけません．そして，その名前にはBASICで書かれたプログラムと区別するために，"mem"という拡張子をつけておくことにします．まず作るファイルの名前を聞いてくるようにしましょう．拡張子は，ストリングの結合を使って付け加えます．

```
INPUT "FILE NAME" ; NM$ :
NM$+ "mem"
```

次にディスクに書くメモリの最初のアドレスと最後のアドレスを入力するわけですが，モニタでは16進で入力しました．N-BASICでは，16進数を表す場合，前に"&h"を付けて入力しなければなりません．

いちいちこんなものを付けて入力するのは面倒なので，まず文字型変数で入力して，その前に"&h"をプログラムで付け加えるようにします．たとえば，次のようにします．

```
INPUT "START ADDRESS" ; SA$
SA=VAL ("&h"+SA$)
IF  SA <0  THEN  SA=SA+65536!
```

このプログラムでは，まずスタートアドレスを文字変数SA$に読み込み，"&h"を前に付けて，VAL関数を使って数値に変換していますが，POKEコマンドと，PEEK関数の引数が負数を受け付けませんので，65536を加えて正数にしています．この3行の命令並びは16進数を入力するときの基本的なパターンです．もちろん，最後の行などは，IF文以外のビット操作（論理式を使う）によって書き変えることもできます．たとえば

```
SA=SA  AND  &HFFFF
```

でもよいわけです．

さていよいよディスクに書くわけですが，まずOPENで，書き込みファイルを用意します．

```
OPEN  NM$  FOR  OUTPUT  AS  #1
```

これで，NM$という名前の出力ファイル#1が用意できました．まずスタートアドレスとエンドアドレスを書き込みます．

```
PRINT#1, SA, EA
```

次に，メモリの内容をPEEK関数で読み出して，書き込みます．

```
FOR  AD=SA  TO  EA
    PRINT#1, PEEK (AD)
NEXT  AD
```

ここで，ADはこのFOR-NEXTループの制御変数です．

ファイルへの書き込みが終ったら，それをDisk BASICに知らせるために，ファイルをクローズします．

```
CLOSE#1
```

これで，#1のファイルの使用を終ったわけです．END文によっても，ファイルはクローズされますが，今の場合のように，ファイルを1回だけ使用する場合には混乱は起こりませんが，何度も同じファイルを使って効率よくファイル領域を使用する場合には，たいへんなことになりま

す．OPEN したファイルは，用が済んだら，かならず CLOSE するようにしましょう．

以上をまとめてみますと次のようなプログラムになります．

```
100 '
110 ' save memory image
120 '
130 INPUT"file name";NM$
140 NM$=NM$+".mem"
150 INPUT"start address";SA$
160 SA=VAL("&h"+SA$)
170 SA=SA AND &HFFFF
180 INPUT"end   address";EA$
190 EA=VAL("&h"+EA$)
200 EA=EA AND &HFFFF
210 '
220 OPEN NM$ FOR OUTPUT AS#1
230 PRINT#1,SA,EA
240 FOR AD=SA TO EA
250    PRINT#1,PEEK(AD)
260 NEXT AD
270 CLOSE #1
280 END
```

実際にこのプログラムを使ってみましょう．まず，MON コマンドで，モニタに制御を移します．ここでは，機械語プログラムではなくて，00から 0F (16進) までの数をアドレス C000 (16進) からC 00 F (16進) までのメモリに書き込んでみます．

```
 *SC000
C000 FF-00 FF-01 FF-02 FF-03
C004 FF-04 FF-05 FF-06 FF-07
C008 FF-08 FF-09 FF-0A FF-0B
C00C FF-0C FF-0D FF-0E FF-0F
C010 FF-
*
Ok
```

注意することは，N-BASIC がこの領域を使用していないことです．まだ，何も行なっていないなら，CLEAR 文でN-BASIC の使用限度を指定する必要があります．たとえば，C000番地からユーザーが使用したいのであれば，

CLEAR　　100,　&h BFFF

を実行すればよいのです．ただし，第一引数の 100 は文字領域の大きさを示します．
これを，怠るとN-BASIC によってせっかくのデータがこわされます．次にこのプログラムを走らせます．

```
run
file name? test
start address? c000
end   address? c00f
Ok
```

これで，アドレス C000 (16進) から C00F (16進) までのメモリの内容が入った，"TEST.mem" というファイルができたことになります．

それでは次に機械語用のロードプログラムを作ってみましょう．どの番地からロードするかは，ファイルに記録されているので，ファイル名だけを入力すればいいのです．

INPUT "FILE　NAME" ; NM$
NM$＝NM$＋ "mem"

ファイルを読み込む場合にも，書き込みの場合と同じようにまず，ファイルをオープンする必要があります．

OPEN　NM$　FOR　INPUT　AS　#1

これで，NM$ という名前の入力ファイル#1が用意できました．まずスタートアドレスとエンドアドレスを読み込みます．

INPUT#1,　SA,　EA

どの番地に書き込むつもりなのか表示させましょう．このとき16進形式で出力する方が見やすいので，HEX$ とRIGHT$ 関数を使って16進に変換します．

SA$＝RIGHT$ ("000"＋HEX$ (SA),　4)

この文によって，数値変数SA (スタートアドレスが入っている) は，16進形式の文字型変数 SA$ に変換されます．スタートアドレスとエンドアドレスを表示したのち，実際にメモリに書き込み

ます．

```
FOR AD=SA TO EA
    INPUT#1, D : POKE AD, D
NEXT AD
```

変数ADは，FOR-NEXTループの制御変数で，今書き込むアドレスを指しています．変数Dは，INPUT文が関数でなくて，ステートメントであるために使用した，一時使用の変数です．ファイルの読み込みが終ったらクローズしておきます．

```
CLOSE#1
```

これで今使用した，入力ファイル#1の使用が終ります．

以上をまとめてみますと次のようになります．

```
100 '
110 ' load memory image
120 '
130 INPUT"file name";NM$
140 NM$=NM$+".mem"
150 OPEN NM$ FOR INPUT AS #1
160 INPUT#1,SA,EA
170 SA$=RIGHT$("000"+HEX$(SA),4)
180 EA$=RIGHT$("000"+HEX$(EA),4)
190 PRINT "start address",SA$
200 PRINT "end   address",EA$
210 '
220 FOR AD=SA TO EA
230   INPUT#1,D:POKE AD,D
240 NEXT AD
250 CLOSE #1
260 END
```

実際にこのプログラムを使う前に，あらかじめメモリの内容を壊しておけば，ロードルーチンの働きが，わかります．たとえば，0FF(16進)を書き込んでおきましょう．

```
Ok
mon
*SC000
C000 00-FF 01-FF 02-FF 03-FF
C004 04-FF 05-FF 06-FF 07-FF
C008 08-FF 09-FF 0A-FF 0B-FF
C00C 0C-FF 0D-FF 0E-FF 0F-FF
C010 FF-FF FF-FF FF-FF FF-FF
C014 FF-FF FF-FF FF-FF FF-FF
C018 FF-
*
```

ロードプログラムを走らせます．

```
run
file name? test
start address C000
end   address C00F
Ok
```

うまくメモリに書き込まれているか，モニタコマンドを使ってためしてみます．

```
Ok
mon
*SC000
C000 00-   01-   02-   03-
C004 04-   05-   06-   07-
C008 08-   09-   0A-   0B-
C00C 0C-   0D-   0E-   0F-
C010 FF-   FF-   FF-   FF-
C014 FF-   FF-   FF-   FF-
C018 FF-
*
Ok
```

正確にロードされたことがわかります．

ここでどのようにデータがファイルに入っているかを知るために，前に作ったファイルダンププログラム(ここではfdumpという名前を付けてある）で見てみましょう．

```
run"fdump"
file name? test.mem
0000 2D 31 36 33 38 34 20 20   -16384
0008 20 20 20 20 20 20 2D 31         -1
0010 36 33 36 39 20 0D 0A 20   6369 ..
0018 30 20 0D 0A 20 31 20 0D   0 .. 1 .
0020 0A 20 32 20 0D 0A 20 33   . 2 .. 3
0028 20 0D 0A 20 34 20 0D 0A    .. 4 ..
0030 20 35 20 0D 0A 20 36 20    5 .. 6
0038 0D 0A 20 37 20 0D 0A 20   .. 7 ..
0040 38 20 0D 0A 20 39 20 0D   8 .. 9 .
0048 0A 20 31 30 20 0D 0A 20   . 10 ..
0050 31 31 20 0D 0A 20 31 32   11 .. 12
0058 20 0D 0A 20 31 33 20 0D    .. 13 .
0060 0A 20 31 34 20 0D 0A 20   . 14 ..
0068 31 35 20 0D 0A            15 ..
Ok
```

おや，たった16バイトのデータをセーブしただけなのにずいぶん量がありますね．ここで前に説明した，「ファイルへの書き込みは文字列として出力される」ことを思い出してください．つまり数値をファイルに書き込むと，それをPRINT文で，プリントしたのと同じだけの文字数が必要になるのです．今の場合，アドレスは2バイト，データはそれぞれ1バイトあれば，表現できるのですから，これはたいへんなムダをしていることになります．それで，Disk BASICで

は，数値の内部表現をそのまま，読み書きするための関数が役に立ちます．アドレスを示している整数変数は MKI$ によって2バイトの文字列に変換できますし，その逆は，CVI 関数によって行います．メモリの内容もこの2つの関数を使って2バイトの情報として読み書きできますが，1バイトのデータを2バイトにして記録するのはムダです．1バイトの数値文字変換関数は用意されていませんが，この変換は CHR$ と ASC 関数によって行なえるのです．

すなわち，CHR$ が1バイトの数値を1文字に変換する関数であり，ASC が1文字を1バイトの数値に変換する関数なのです．これで必要最少の容量で，データを記録することができるのです．

```
100 '
110 ' save memory image
120 '
130 INPUT"file name";NM$
140 NM$=NM$+".mem"
150 INPUT"start address";SA$
160 SA=VAL("&h"+SA$)
170 SA$=MKI$(SA)
180 SA=SA AND &HFFFF
190 INPUT"end   address";EA$
200 EA=VAL("&h"+EA$)
210 EA$=MKI$(EA)
220 EA=EA AND &HFFFF
230 '
240 OPEN NM$ FOR OUTPUT AS#1
250 PRINT#1,SA$;EA$;
260 FOR AD=SA TO EA
270   PRINT#1,CHR$(PEEK(AD));
280 NEXT AD
290 PRINT #1,
300 CLOSE #1
310 END
```

```
100 '
110 ' load memory image
120 '
130 INPUT"file name";NM$
140 NM$=NM$+".mem"
150 OPEN NM$ FOR INPUT AS #1
160 SA=CVI(INPUT$(2,1))
170 EA=CVI(INPUT$(2,1))
180 SA$=RIGHT$("000"+HEX$(SA),4)
190 EA$=RIGHT$("000"+HEX$(EA),4)
200 PRINT "start address",SA$
210 PRINT "end   address",EA$
220 '
230 FOR AD=SA TO EA
240   POKE AD,ASC(INPUT$(1,1))
250 NEXT AD
260 CLOSE #1
270 END
```

前と同じようにして，メモリの内容をセーブして，再び入力してみます．

```
run
file name? test
start address? c000
end   address? c00f
Ok
```

これらのプログラムは，ファイルの中にどのようにデータが入っているか，〝fdump〟プログラムを使って見てみましょう．

```
run"fdump"
file name? test.mem
0000 00 C0 0F C0 00 01 02 03  .ﾀ.ﾀ....
0008 04 05 06 07 08 09 0A 0B  ........
0010 0C 0D 0E 0F 0D 0A        ......
Ok
```

どうです，びっしりデータがつまっているでしょう．このときに注意してほしいのは，

PRINT#1, SA$; EA$;
や
PRINT#1, CHR$ (PEEK (AD)) ;

などの中にある〝;〟です．これがないと間に空白が入ってしまって思うようにデータが，記録されません．ところが，このようにデータをビッシリ詰めてしまうと，前のように INPUT 文ではデータをうまく読み込めません．それは，データとデータの切れ目が分からなくなってしまうからで，Disk BASIC はデータをどんどん読み込んで，すべてのデータが，文字変数 SA$ に入ってしまうか，あるいは，255文字を越える場合にはエラーになってしまいます．前のプログラムではデータとデータの切れ目にキャリッジリターンとラインフィードのコード（0Dと0A）が入って，データの切れ目が分かったのです．そこで，INPUT$ 関数を使って必要な文字数だけ読み込むようにしたわけです．

このプログラムでの教訓は，「数値をファイルに書き込むときには内部表現を使え．」ということです．数値を内部表現に対応する文字列に変換する関数は，ランダムアクセスファイルでは，

必ず必要なものですが，シーケンシャルファイルでも，使わなければディスクのむだ使いをしていることになるのです．

Disk　BASIC では，シーケンシャルファイルの終りを EOF コード（1A（16進））で判断しています．したがってデータの最後のデータが EOF コードである場合には，そのデータが読まれなくなってしまいます．そこで，このプログラムでは最後に余分の CR／LF コードを送っています．

4．3　音声発生

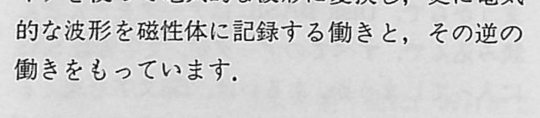
最近は少々下火になりましたが，一時はマイコンで音声を発生させることが流行しました．

米国のAというパーソナルコンピュータの場合に至っては，何とハードウェアを追加しなくても，ソフトウェアだけで声を出してくれます．

他のパーソナルコンピュータでできる事ならば，PC-8001でも不可能なはずはないと考えられます．そこで，PC-8001を使って音声を発生できるかどうか実験してみました．

4．3．1　原理

音声を扱う代書的な家電製品として，テープレコーダーがあります．これは音声の波形をマイクを使って電気的な波形に変換し，更に電気的な波形を磁性体に記録する働きと，その逆の働きをもっています．



コンピュータで音声を発生させる方法も，基本的にはテープレコーダと全く同じです．電気的に変換された音声の波形をコンピュータが記憶するわけです．

しかし，テープレコーダの場合は，入力の音声波形をアナログ量のままテープの磁性体上に記録しますが，マイコンの場合（デジタル機器はすべてですが）アナログ量を直接取り扱うことができません．そこでアナログ量をデジタル量に変換する回路が必要になります．

デジタル化もいろいろな種類があります．図1を見て下さい．

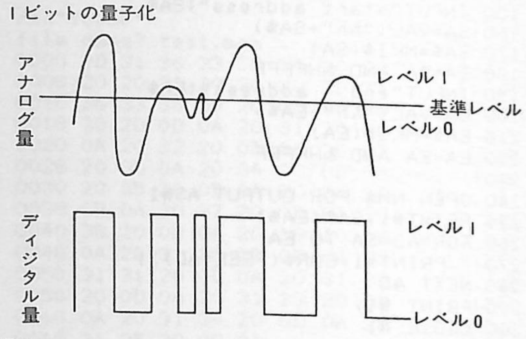
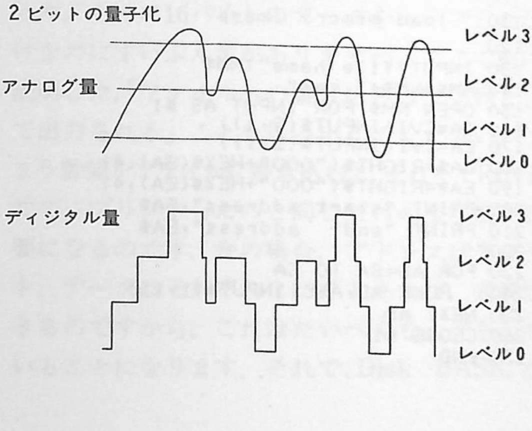


図1a



アナログ量を，1ビットのデジタル量に変換する場合，図 1a のように基準レベルを定めておき，アナログ量が基準レベル以上のときデジタル量の1に，基準レベル以下の時にはデジタル量の0に対応させればよいわけです.

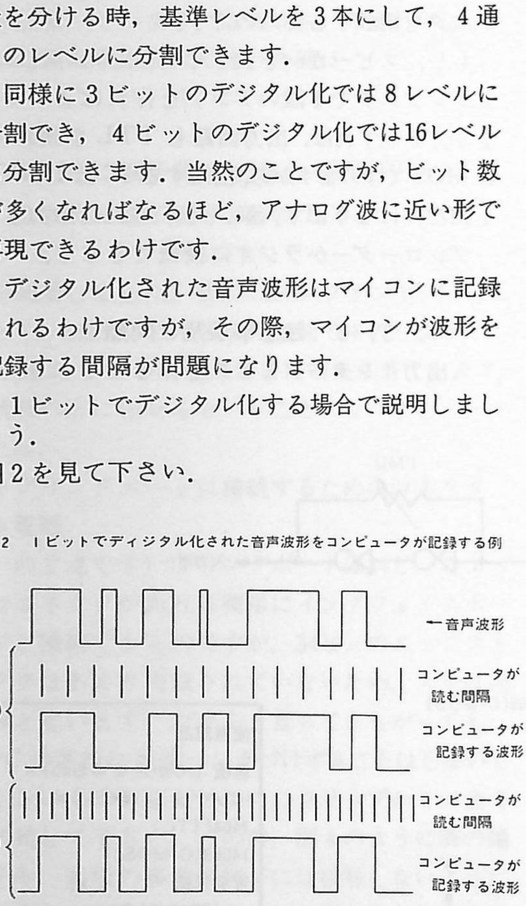
2ビットのデジタル化の場合には，デジタルでも4段階のレベルを表せますから，アナログ量を分ける時，基準レベルを3本にして，4通りのレベルに分割できます.

同様に3ビットのデジタル化では8レベルに分割でき，4ビットのデジタル化では16レベルに分割できます. 当然のことですが，ビット数が多くなればなるほど，アナログ波に近い形で再現できるわけです.

デジタル化された音声波形はマイコンに記録されるわけですが，その際，マイコンが波形を記録する間隔が問題になります.

1ビットでデジタル化する場合で説明しましょう.
図2を見て下さい.

図2　1ビットでディジタル化された音声波形をコンピュータが記録する例



デジタル化された音声波形は一定周期でコンピュータが読むわけですが，コンピュータが読む間隔（サンプリング周期）によって，実際にコンピュータに記録される波形と，デジタル化された音声波形とに違いが生じてくるわけです.

図.2では，aがbの半分の周期で音声波形をサンプリングしています. 記録される波形はaよりもbのほうが明らかに音声波形に近いことが分ると思います.

ここまでの説明でコンピュータで音声波形を扱う時，できるだけ実際の音声波形に近づける方法は次の通りになることが分ってもらえたと思います.

1)デジタル化の際のビット数を多くする.
2)サンプリング周波数を高くする.

上の2つの条件を満たせば音声を再生する時の音質は向上するわけですが，逆に条件を満たせば，コンピュータが記録する情報量が増加することになります. コンピュータのメモリには限度がありますから，音質を適当な所で切り捨てて，記録する時間を得ることになります.

例えば，1ビットのデジタル化で，サンプリング周波数8kHzで,12秒記録できたとすると，2ビットのデジタル化で，サンプリング周波数16kHzの場合，3秒しか記録できません.

サンプリング周波数を下げて時間を稼ぐといっても，音声を再生するのに最低限必要な周波数というものがありまして，あまり下げることはできません.

これは情報理論の標本化定理により，周波数Fの信号をサンプリングして再生するためには，サンプリング周波数は2×F 以上必要になります. 従って，もしサンプリング周波数を8kHzにすれば，音声信号は4kHz以下のものしか再生できません.

4．3．2　実験回路

今回はPCが〝人間語〟を話せるようにするため，最も簡単なハードウェアを試みました. 方法はこれ以外にも多数ありますが，とにかく簡単で,PC本体に影響を及ぼさないようにしたわけです.

ハードウェアをできるだけ簡単にするため，デジタル化は１ビットで行いました．また，サンプリング周波数はソフトウェアで選べるようにしてあります．

さて，難しい話はぬきにして回路図を図３に示します．注意事項及び問題点を読んでから作ってみて下さい．

一応回路について説明しますが，回路に興味のない人は読まなくてもかまいません．

まず入力回路です．入力回路はラジオやテープレコーダーのイヤホン端子から入力された音声波形を，C-MOS のインバータを使用して，１ビットのデジタル波形に変換します．初段のインバータに帰還抵抗をつけ，アナログアンプとして使用するわけです．次段のインバータ２個に帰還抵抗をつけたものは，シュミットトリガで，アナログアンプで増幅した波形をデジタル化しています．

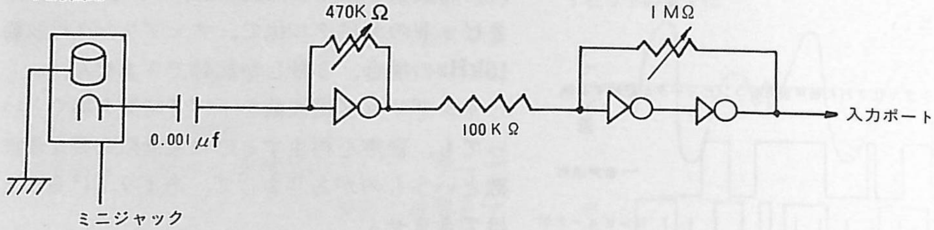
出力回路は，できるだけ簡単にするため，トランジスタ等は使わず，TTL のインバータで直接スピーカーを鳴らしてしまいます．電流を得るため，インバータを２個並列にしました．尚，コンデンサと抵抗も並列にしたのはローパスフィルタを構成するためのです．

もし，スピーカーを大きく鳴らしたければ，トランジスタで１段のアンプを作ればよいでしょう．そうすれば，出力回路も TTL を使わず，C-MOS ですみますから，使用する IC は C-MOS １個だけになります．又，そうすれば出力もテープレコーダーかラジオに接続できます．

4．3．3　注意事項及び問題点

入出力インターフェイスを作る時２～３問題

入力回路．



インバータは　MC14069(C-MOS)

出力回路



図　3
入出力インターフェイス回路図

インバータは　7404

使用部品
基板（ICがさせるもの）×１
ICソケット（14ピン）×２
7404(TTL)
14069(C-MOS)
可変抵抗
　470KΩ ×２
　100KΩ ×１
　1MΩ ×１
固定抵抗
　1MΩ ×１
　400Ω ぐらい×１
コンデンサ
　0.01μF×１
　0.001μF×１
小型スピーカ×１

がありました．まず第1に，インターフェイスをPC-8001のどこに接続するかについてです．PCから外部へ出ている外部バスにポートを接続しようかと考えたのですが，回路が複雑になりそうなのでやめにして，結局プリンタ用のポートを使用することにしました．プリンタ用ポートの端子構成はPC-8001のユーザーズマニュアルに載っています．このうち，入出力用に1本づつ使用するわけです．今回は

入力用に端子番号21番の$\overline{\text{READY}}$

出力用に端子番号3番のPD0

を使用しました．更にGND(グランド)を，PCとインターフェイスボード間で接続します．

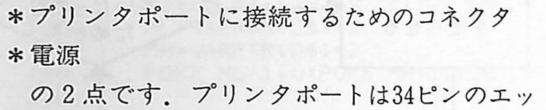
実は，あとの問題点はまだ解決されておらず，読者の方に任せるということになりました．それは，

＊プリンタポートに接続するためのコネクタ

＊電源

の2点です．プリンタポートは34ピンのエッジコネクタがあれば簡単にインタフェイスボードと接続できるのですが，34ピンのエッジコネクタはあまり市販されていないため，入手は困難と思います．だからと言ってまちがっても，PCの基板に直接ハンダづけすることはしないで下さい．できるだけワニロクリップかミノ虫で接続して下さい．この時，図4のように隣の端子や，基板の裏側の端子には接触しないように注意が必要です．できれば必要な端子以外はテープで覆ってしまえばよいと思います．

図4



正面から見た図



上から見た図

3番→インターフェイス基板の出力回路へ接続
21番→インターフェイス基板の入力回路へ接続
31番→インターフェイス基板のGNDへ接続

これ以外はテープで覆って接触しないようにする

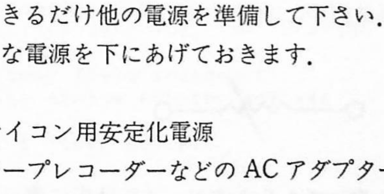
もう1つの読者の方に任せる問題点は電源です．これはインターフェイス基板上のICに供給して，スピーカーを鳴らすための5Vの電源が必要なのです．実はPCのプリンタポート中にも"HIGHT"という端子があります．テスターで測定してみると確かに5Vあるので，最初はそこから取ることを考えたのですが，残念ながら今回の回路ではダメでした．
自信のある人はPCの基板内のどこかから5Vを取り出せばよいのですが，初心者には勧めません．できるだけ他の電源を準備して下さい．使えそうな電源を下にあげておきます．

1．マイコン用安定化電源

2．テープレコーダーなどのACアダプターで4.5～5Vのもの．

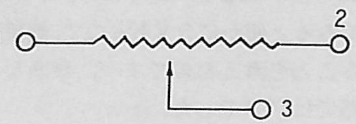
3．最悪の場合：乾電池3本直列でも動くと思います．ただしこの場合は新品の電池を使用して下さい．

次に初心者むけに注意事項を書いておきましょう．まずICのピン配置についてです．今回使用するICは，C-MOSタイプとTTLタイプのインバータです．この2つのICはタイプは違いますが，ピン配置は同じです．



尚 IC は基板に直接ハンダづけはしないで IC ソケットを使用して下さい．又 C-MOS IC は静電気に弱いので回路が完成するまではアルミホイルのような導電性のものにつつんでおくようにして下さい．TTL-IC の場合はそこまでしなくてもかまいませんが，やはりソケットに装着するのはハンダづけを終えてからのほうがいいでしょう．

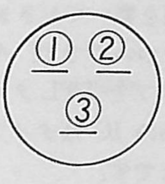
もう1つ，可変抵抗について説明しておきましょう．可変抵抗は一般に3本の足で構成されています．各足の機能は下の通りです．



回路図中には



と書いてありますが，これは先に書いた図の1と3又は2と3を使って構成することになります．実際の可変抵抗器の足は3角形になっていて下のように対応しています．



4．3．4　回路が完成したら

インターフェイスが完成したら一応チェックしなければなりません．まず IC を装着する前に，総ての可変抵抗を最大にしておいて下さい．この時，可変抵抗器の足の1と3を使ったか，2と3を使ったかで，最大値にするために可変抵抗を回す方向が異なりますから注意が必要です．

次は出力回路のチェックです．図6のフローチャートとテストプログラムでチェックして下さい．

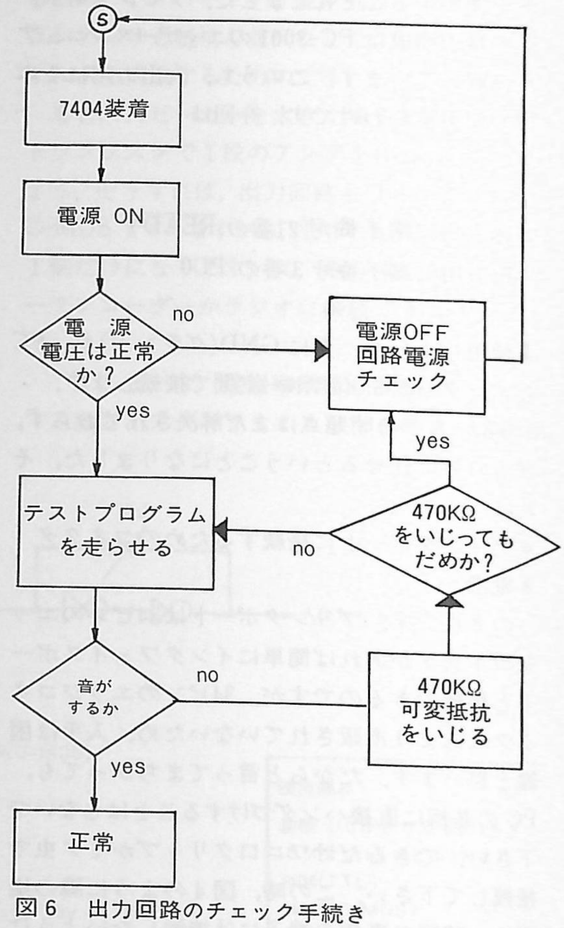


図6　出力回路のチェック手続き

テストプログラム

```
10  FOR M=1 TO 1000
20  OUT  &H10,0
30  OUT  &H10,1
40  NEXT  M
50  END
```

さて，出力回路で音が出る場合には，次は入力回路のチェックをするのです．このチェックは

少々面倒なので，まずC-MOS　ICを装着して電源が正常なのを確認したら，次のプログラムの項を読んで，プログラムを入力して下さい．実際にプログラムを動作させながら，試行錯誤で調整するわけです．

音声を入力して(hコマンド)記録し，それを再生して(tコマンド)みて，その音によって調整します．

1．もし音がでないならば，
　100KΩと1MΩの可変抵抗を下げる．
2．もし何も入力しなくても雑音ばかりならば，
　100HΩと1MΩの可変抵抗を上げる．

を繰り返します．2個の可変抵抗をどう調整しても全く音がでないとか，雑音ばかりならもう1度回路の配線をチェックして下さい．

尚，470KΩの可変抵抗はあまりいじらない方が良いと思います．もしこの抵抗値を零にしたりすると，C-MOSのインバータが壊れかねません．

調整の結果，音声を入力した時には音がして，何も入力しない時には静かなままという状態になったら，今度は〝音〟を注意深く聴いて下さい．きっと雑音の中におぼろげにあなたの声がきこえていると思います．

4．3．5　プログラム

さて，作製した回路を使って音声の入出力をするわけですが，次のような簡単なプログラムを作ってみました．

```
100 '*
110 '*    Demo program for Section 4.3
120 '*
130     CLEAR 300,&H9FFF                          'reserve memory area
140     DEFUSR1=&HA000:DEFUSR2=&HA003             'define user function location
150     FIELD #1,128 AS A$,128 AS B$
160     FIELD #1,128 AS C$,128 AS D$              'dummy field statement
170     ML=VARPTR(C$)+2                           'set string pointer address
180     MH=VARPTR(D$)+2
190     POKE ML-1,0:POKE MH-1,128                 'set lower address
200 '
210     FOR I=&HA000 TO &HA03C
220        READ D:POKE I,D                        'read machine language program
230     NEXT I
240     GOSUB 310
250 '
260     INPUT "COMMAND:";COM$
270       P=INSTR("dthsle",COM$)
280       ON P GOSUB 310,370,420,490,630,910
290     GOTO 260
300 '*
310 '*    Delay set
320 '*
330     INPUT "DELAY COUNT:";DY
340     POKE &HA032,DY:POKE &HA038,DY+2           'set delay count
350    RETURN
360 '*
370 '*    Talk
380 '*
390     A=USR1(0)                                 'USR1 is talk routine
400    RETURN
410 '*
420 '*    Hear
430 '*
440     PRINT "SPEAK NOW!!"
450     A=USR2(0)                                 'USR2 is hear routine
460     BEEP
470    RETURN
```

```
480 '*
490 '*     Save
500 '*
510     PRINT "save routine"
520     INPUT "FILE NAME:";NM$                       'input file name
530     OPEN NM$ AS #1
540       FOR P=0 TO 31
550         L=&HA0+P
560         POKE ML,L:POKE MH,L                       'set save address
570         LSET A$=C$:LSET B$=D$                     'move from memory to buf. #1
580         PUT #1                                    'write 256 byte
590       NEXT P                                      'save &HA000 - &HC000
600     CLOSE #1
610   RETURN
620 '*
630 '*     Load
640 '*
650     PRINT "load routine"
660     INPUT "FILE NAME:";NM$                       'input file name
670     OPEN NM$ AS #1
680       FOR P=0 TO 31
690         GET #1                                    'read 256 byte
700         L=&HA0+P                                  'L=high order of load address
710         POKE ML,L:POKE MH,L                       'set load address
720         LSET C$=A$:LSET D$=B$                     'move from buf.1 to memory
730       NEXT P                                      'load &HA000 - &HC000
740     CLOSE #1
750   RETURN
760 '*
770 '*   Machine language program data
780 '*
790   DATA &Hc3,&H1b,&Ha0,&H21,&H40,&Ha0
800   DATA &H0e,&H08,&Hdb,&H40,&H1f,&Hcb
810   DATA &H18,&Hcd,&H31,&Ha0,&H0d,&H20
820   DATA &Hf5,&H70,&H23,&H7c,&Hfe,&HC0
830   DATA &H20,&Hec,&Hc9,&H21,&H40,&Ha0
840   DATA &H0e,&H08,&H7e,&Hd3,&H10,&Hcd
850   DATA &H37,&Ha0,&H1f,&H0d,&H20,&Hf7
860   DATA &H23,&H7c,&Hfe,&Hc0,&H20,&Hee
870   DATA &Hc9,&H16,&H00,&H15,&H20,&Hfd
880   DATA &Hc9,&H16,&H00,&H15,&H20,&Hfd
890   DATA &Hc9
900 '
910   END
```

このプログラムは，全部で6つのコマンドを持っています．hは音声を入力してそのデータをメモリ中にしまうコマンド，tはメモリ中の音声データを出力するコマンドです．また，メモリ中のデータをディスクにしまうことのできるようにs (SAVE) 及びl (LOAD) が用意してあります．サンプリング周期を自由に変更できるようにd (DELAY) も準備してあります．プログラムについて順を追って説明していきましょう．

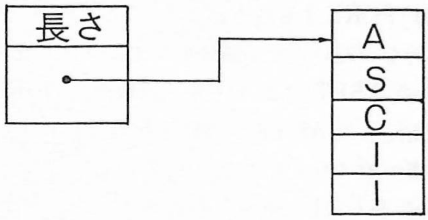
120行の CLEAR 文では BASIC で使用するメモリの上限を 9FFF番地までに制限します．これは，A0000~C000番地までの8K分のメモリを機械語ルーチン及び音声データの格納用に使用するためです．140行では機械語ルーチンを使用するためにその先頭番地の定義をしておきます．A000番地からは出力のルーチン，A003番地からは入力のルーチンとして使用しています．150行の FIELD 文は，ディスクとのデータのやりとりを GET, PUT で行なうためのものです．

160行の FIELD 文はダミーで C$ と D$ に128文字分の大きさを与える目的のみで使用して

います．ここで，メモリ中のデータのロード，セーブを行なう方法について説明しておきましょう．実行したい仕事としては７・２節で説明した機械語ファイルのロード，セーブと同じなのですが，通常の方法では８Kバイト分のロード，セーブに１分程度の時間がかかってしまうようなので，ここで特別な方法を使ってロード，セーブを行なっています．

まずセーブの方法について説明しましょう．基本的には PUT を使用しているので，メモリ中のデータを256バイトずつランダムバッファに持ってきては PUT することによってセーブが実現されています．ここで問題となるのが，どのような方法でバッファへのデータの転送を行なうかです．PEEK，POKE を使用して FOR-NEXT ループで繰り返したところでは，この転送にかなりの時間が消費されるためにセーブに１分程度の時間がかかってしまったのです．そこで，ここでは文字変数と LSET 文を使って転送を行なう方法を考えてみました．３章でも説明したように，LSET 文を使えば文字列データの移動が生じるという性質を利用したのです．150行の FIELD 文では A$ と B$ が１番のランダムバッファ上にとられていますから，A$ 及び B$にＬSET 文で文字列の代入を行なえば，バッファへのデータの転送が実現されることになります．問題は代入する文字列が，メモリ中のデータを表現するようにしてやることだけです．幸い N-BASIC には，VARPTR 関数があり，変数の値の操作等ができるようになっているので，これを利用して，メモリ中のデータを文字変数で表わすようにしています．ロード時は同様にして，メモリ中を示している文字列に対して GET で読み込んだ文字列を LSET 文により代入することによって実現します．160行に表われる C$ 及び D$ はこの目的で使用するための文字変数です．

この先の説明をするには文字変数が BASIC 内部でどのように表現されているかを知っていなければなりません(詳しくは，PC#1参照)．まず，BASIC の変数領域には，文字列の長さと，実際に文字列データの入っているアドレスを示すポインタとが入っています．



VARPTR が返す値は，実際に文字列の入っているアドレスではなく，長さとポインタの入っているアドレスになります．実際には，VARPTR の返す番地には，文字列の大きさ，その次の番地には，実際の文字列データの入っている番地の下倍バイト，続いて上位バイトが入っています．ですから，170行では，C$ の示す文字列データの入っている番地の上位バイトを保持する番地が ML に得られます．同様にして，180行には MH に D$ のそれが得られます．190行では POKE 文により C$ 及び D$ の示すアドレスの下位を０及び128に設定しておきます．

210行から230行のループでは DATA 文で用意した機械語ルーチンを読みA000番地以降に書き込みます．このようにして機械語ルーチンも BASIC で準備することができるのです．

240行では，DELAY　COUNT の初期設定を行ないます．260行から290行はこのプログラムのメインドライバに当るループで，コマンドを入力し，各処理ルーチンへ ON-GOSUB 文で分岐します．

310行からは DELAY　COUNT の設定，すなわち d コマンドの処理ルーチンです．通常このカウントは10から20で良いでしょう．10で約

4秒，20で約8秒の間，音声の入出力が行なえます．

490行からは S コマンドの処理ルーチンです．A000番地からC000番地までの8Kバイトを，256バイトずつ PUT するために，FOR-NEXT で32回ループを回します．550行では PUT すべきメモリアドレスの上位バイトを求め，560行でその値を POKE することにより，C$ と D$ が転送を行ないたいメモリ領域をさすようにします．570行の LSET によりメモリ領域から FIELD 文で指定した A$, B$ への転送が行なわれ PUT の準備が整のうことになります．

630行からはLコマンドの処理ルーチンです．ロードの手続きもセーブの手続きと，きわめて良く似ています．まず GET により A$，B$ にデータが読み込まれるので，これを LSET により，メモリ領域をさすC$，D$ に代入することによって転送を行なうのです．

790行からのDATA 文の内容は機械語のプログラムです．データは全て16進数で書いてありますから，機械語を御存知ならば容易にわかることと思います．

4．3．6 まとめ

今回はとにかく簡単に話すことを目的としたため，多分に音質が犠性になってしまったようですが，とにかく我々の実験では PC は人間語を憶え，話せるようになりました．簡単なインターフェイスなので読者の方も1度試してみて下さい．腕に自信のある人は回路を改良して音質を向上させるのもおもしろいと思います．

参考　アスキー　1979年2月号　特集　マイクロコンピュータと音声

23ページ　ソフトウェアだけでも音声の録音　再生が十分に可能！

4．4 「辞書」-ランダムアクセスのまとめ-

ランダムアクセスファイルのまとめとして，「辞書」プログラムを紹介します．このプログラムは，実用的な辞書としては，ムダがあるのですが，3章で解説した基本的テクニックが実際のプログラム中で，どのように使われるか理解するのには，よい例だと思います．

このプログラムには，単語とレコード番号を結びつける対応表，調べたい単語を高速で探し出す二分検策，対応表への挿入と削除，未使用レコードのスタックによる再利用などが応用されています．

このプログラムでは3つのファイルを使っています．1つは，〝dictio. dat〟です．このファイルは，単語の意味が記録される，ランダムファイルです．1つの単語につき，1レコードを使用しています．2番目のファイルは〝dictio. idx〟という名前で，単語からレコード番号を引き出すための，対応表が記録されている，シーケンシャルファイルです．3番目のファイルは〝dictio. fre〟という名前で，未使用レコード番号が記録されている，シーケンシャルファイルです．

4．4．1 使い方

初めてこのプログラムを走らせる場合には，2つのシーケンシャルファイルをあらかじめ，作っておく必要があります．作るといっても，ディレクトリにファイル名が載っていればよいのであって，内容は無くてもかまいません．したがって，まず次の命令列を実行します．

```
open "dictio.idx" as #1
Ok
close
Ok
open "dictio.fre" as #1
Ok
close
Ok
```

これで準備はできました．では，RUN しまし

ょう．最初にコマンドを聞いてきます．コマンドには6種類あります．

1．append（追加コマンド）
2．delete　（削除コマンド）
3．list　　（一覧コマンド）
4．search　（検策コマンド）
5．instring（部分列検策コマンド）
6．exit　　（終了コマンド）

それぞれのコマンドは頭の1文字のみ打ち込むことによって動作します．この場合にキャリッジリターンは不用です．頭の1文字のみ入力すれば，後の文字は自動的に表示されます．例えば，〝A″と入力するだけで，〝append new key word″と表示して，追加モードになります．未定義の文字を入力すると，BEEPして，エラーを知らせます．この場合には音が出るだけでCRT上には何も表示されません．次に各コマンドの説明をしましょう．

appendコマンド：このコマンドは，新しい単語と，その意味を辞書に追加するためのコマンドで，〝append：？″と聞いてきますから，追加したい単語を入力します．次に〝meaning：″と聞いてきますから，その単語の意味を入力します．意味を入力するときに注意することは，キャリッジリターンも入力することができることです．こうすると，意味を番号順に頭をそろえて入力するようなことができて，とても見やすくなります．

```
run
command:append new key word

append:? book
meaning:
n.ホン

append:? pencil
meaning:
n.エンピツ
append:?
end of append

command:
```

すでに登録してある単語を，appendで再び指定すると，新しく入力した意味がその単語の意味となって前の意味は失われます．

appendモードからぬけ出して，commandモードに戻るためには，〝append？″の後，キャリッジリターンのみ入力します．以下のコマンドも，それぞれのモードから，commandモードに戻るためには，すべてキャリッジリターンのみ入力すればよいようになっています．

deleteコマンド：このコマンドは，すでに登録してある単語を削除するのに使います．〝delete：？″と聞いてくるので，削除したい単語を入力します．そうすると，対応表から入力された単語を削除し，その単語の意味を記録していたランダムファイルを，後から追加する単話のために解放します．もし入力された単語が対応表に載ってなければ〝not found″と表示して，再び〝delete：？″と聞いてきます．

```
command:delete key word

delete:? book

delete:?
end of delete
```

list　コマンド：このコマンドは，現在登録されているすべての単語とその単語の意味が記録されているレコード番号，最後に総数とランダムファイルの総レコード数を表示します．

```
command:list up key words

paper        3
pencil       2

number of key words: 2
number of records: 3
```

searchコマンド：このコマンドは，辞書の本来の働き，単語の意味を表示するものです．〝search：？″と聞いてきますから，意味を調べたい単語を入力します．対応表の中を探してその単語が見つかれば，意味をディスクから読み出して表示します．もし対応表の中にその単

語が見つからなければ，〝not found〟と表示します．

```
command:search key word
search:? book
n.      ホン
```

instring　コマンド：search コマンドでは，単語のスペルが完全に一致しないと，意味が調べられません．スペルを完全に覚えていないとか，名詞は載っているが形容詞は載っていないなど，単語の頭から数文字目まで調べることができると便利です．そのために用意されたコマンドが，instring search，部分文字列検策です．〝instring：？〟と聞いてきますから，調べたい単語の前から一部分を入力します．そうすると，入力された文字列で始まる単語がキャリッジリターンを入力するたびに1つずつ，アルファベット順に表示されます．目的の単語が表示されたら〝y〟を押してください，その意味が表示されます．入力された文字列で始まる単語が対応表になくなってしまえば〝not found〟と表示します．

```
command:instring search

instring:? ac
accept
access
accident
accompany
accomplish
vt.ナシトゲル,カンセイスル.<モクテキ ヲ>ハタス(carry out)

instring:?
Break in 2540
Ok


instring:? ap
apart
appeal
appear
appearance
apple
appoint
approach
apron
not found

instring:?
end of instring search

command:
```

exit コマンド：このコマンドは，プログラムの終了コマンドです．辞書に変更があれば，変更された対応表と未使用スタックをディスクに書き込み，変更がなければ，すぐに〝Good bye〟と表示してプログラムを終ります．

```
command:exit
Good bye.
Ok
```

4．4．2　プログラムの解説

1030行　文字領域を5000文字分割保します．プログラムは32Kで試したものです．

1040行　このプログラムで使用する数値変数はすべて整数型です．

1050行　変数SIZE は単語の最大数です．

1060行　文字配列 W$（　）は，対応表の名前の部分で，単語がアルファベット順に入ります．

1070行　数値配列R（　）は，対応表のレコード番号の部分で，単語の意味を記録してあるレコード番号が入ります．

1080行　数値配列 FS（　）は，未使用レコードを再利用するためのスタックです．

1090行　特に文字変数を利用するまでもないのですが，見やすくするために2つの変数を導入しました．文字変数 CR$ にはキャリッジリターンコードの13が，文字変数 LF$ にはラインフィードコードの10が代入されます．

1100行　数値変数 FLAG はディスク上のデータに変化があったかどうかを示す論理変数です．論理変数は真1と，偽0の2値を取ります．ディスクに変化がないのに，まったく同じデータ

を書くのはムダです．そもそも，辞書は引くためにあるのですから，一度入力してしまえば，ディスクを読み出すだけで書き込みはしないのが普通です．
こうすれば終了したときに，〝dictio. idx〞 と〝dictio. fre〞の書き込みを省略できます．

1110行～1210行　対応表の読み込みです．数値変数 N が単語の総数で，EOF 関数を使ってファイルが終るまで，単語の数を数えながら入力します．1180行に注目してください．機械語ファイルのところで述べたように，シーケンシャルファイルにおいて，数値を扱う場合 PRINT#と INPUT#で直接数値を入出力すると ASCII 形式となるので，ムダが多くなります．そこで MK シリーズと CV シリーズの関数を使って内部表現の形で出入れをするのです．一般的に次のようになります．
256未満の整数の場合

PRINT#1, CHR$ (X)；で書き込み,
X=ASC (INPUT$ (1 ,#1)) で読み出す．

整数型の場合
PRINT#1, MKI$ (X)；で書き込み,
X=CVI (INPUT$ (2 ,#1)) で読み出す．

単精度型の場合
PRINT#1, MKS$ (X)；で書き込み,
X=CVS (INPUT$ (4 ,#1)) で読み出す．
倍精度型の場合
PRINT#1, RKD$ (X)；で書き込み
X=CVD (INPUT$ (8 ,#1)) で読み出す．

1220行～1310行　未使用スタックの読み込みです．数値変数 FR はスタックポインタです．

1330行～1350行　辞書の中心〝dictio. dat〞をオープンします．数値変数 MAX はファイルの最大レコード数で，初期値は LOF 関数によって代入されます．レコードの区分は前255文字が意味の入る部分で，文字変数 MF$ に割当てられます．最後の1バイトは，意味の有効長が入る部分で，文字変数 L$ に割当てられます．どうしてこのような変数が要るのかというと，シーケンシャルファイルでは文字列をディスクに書き込むと，その文字列が含んでいる文字数だけしか書き込まれませんが，ランダムファイルに書き込む場合は LSET やRSET によって余分な空白を付け加えて，固定長にしているために，読み込んだときに余分な空白も入ってしまいます．出力形式が一定文字数で固定しているような表の場合にはよいのですが，自由書式の場合には空白が目ざわりです．そこで，ランダムファイルに書き込むときに，文字列のもともとの長さを一緒に記録しておこうというわけです．

1360行～1450行　コマンドに応じて，それぞれのサブルーチンに分枝する部分です．コマンド入力の仕方はいろいろあるのですが，ここでは INSTR 関数を使ってみました．1410行でキーボードから1文字入力します．その1文字を命令表〝esaldi〞から INSTR 関数によって探すのですめ．例えば〝E〞が入力されたなら数値変数 CM の値は1，〝S〞が入力されたなら2となります．もし CM の値が0ならば，それは，命令表に載ってなかったことになりますから，BEEP して再び入力待ちになります．

コマンドを入力する場合には，コマンドを数文字たたいてキャリッジリターンするよりも，ボタン一発の方が扱いやすいと思います．また数字で入力する場合には，数字と命令との関係を知っていなければなりませんが，頭文字だけ入力するのならば，覚えやすいでしょう．ただ INPUT$ や INKEY$ を使って一発入力する場

合，何も画面にエコーバックしないので，応答に工夫が必要です．このように１文字コマンドの場合は INSTR 関数が便利です．

1470行～1700行　exit コマンドの処理ルーチンです．〝exit〞と表示して，〝dictio. dat〞をクローズします．もしディスクに変化がないなら，1690行に飛んで〝Good bye〞を表示してプログラムを終えます．ディスクに変更があった場合には新しい対応表と新しい未使用スタックを書き込まなくてはなりません．1500行～1570行が対応表の書き込みです．1550行の MKI$ は前に説明した，対応表の読み出し CVI に対応しています．１項目の切れ目にはカンマ〝,〞を使っています．これは単語の中にはカンマが含まれていないこと，CR／LF では２バイト必要ですが，カンマなら１バイトで済むことによるものです．1580行～1670行が未使用スタックの書き込みです．FOR-NEXT ループは必ず一回まわってしまうので，FR＝０のときには，ループに入らないようにしています．もちろん，これは対応表の書き込みの場合にも起きる可能性がありますが，単語が一つもない辞書など意味がありません．それに対して，未使用レコードが無しというのは，よくあるわけです．1660行は EOF コードを分離するための余分なデータです．

1710行～1840行　search コマンドの処理ルーチンです．INPUT 文で入力する場合，キャリッジリターンのみ入力すると，以前代入された値のままですが，ここでは，キャリッジリターンのみ入力した場合は，そのコマンドの終了とみなしますので，1760行で KY$ にヌルストリングを代入して，1780行で変化していないならば，キャリッジリターンのみ入力したと判断してリターンします．ヌルストリング以外が入力された場合には，入力した文字列を対応表から探します．2110行からのサブルーチンは３章で説明した，２分検策です．もし見つからなかったならば，〝not found〞と表示して再び入力待ちとなります．見つかった場合には，その単語の意味，レコード番号R（M）に入っていますから，それを GET#します．1830行で GET#したデータの中かは必要なデータのみを切り出すのに，LEFT$ 関数を使っています．

1850行～2090行　append コマンドの処理ルーチンです．ヌルストリングの使い方は前とほぼ同じです．1920行～1960行で単語の意味を入力するのですが，ちょっとした工夫が凝らしてあります．それはキャリッジリターンも含めて入力する方法です．といっても CR／LF コードをプログラムでくっつけるのです．文字変数 M$ が入力された意味全体のバッファで，文字変数 M1$ が入力された一行分が入ります．まず，M$ も M1$ もヌルストリングに初期設定しておきます，次に M1$に LINEINPUT で一行分読み込みます．それを M$ に加えて，終りに CR$ と LF$ を付ける，この繰り返しです．ここでも，キャリッジリターンのみ入力した場合を意味入力の終りとしていますから，M1$ に毎回ヌルストリングを代入しています．1980行で２分検索サブルーチンを呼んでいきます．もし見つかれば，レコードの内容のみを変えればよいのですから，2070行へ飛びます．新入りの単語ならば，対応表へ挿入します．対応表が更新されたのですから，変更フラグ　FLAG をセットしておきます．未使用スタックを調べてまだ余りがあるならば，そのレコードを使い，なければファイルを拡張します．2070行でランダムバッファに書き込むのですが，そのときに長さもいっしょに書き込みます．CHR$ 関数と ASC 関数は，MKシリーズと CV シリーズの中にはありませんが，255以下の整数をランダムファイルとやりとりするために使うと，１バイトで済みます．

2100行〜2180行　２分検索サブルーチンです．

このサブルーチンは，３章で説明したものとまったく同じですので，そちらを見てください．

2190〜2300行　list コマンド処理ルーチンです．単語がゼロの時に FOR-NEXT ループに入らないように，2230行に IF 文を入れてあります．

2310〜2470行　delete コマンド処理ルーチンです．変更フラグをセットする以外は３章で説明したものと同じです．

2480行〜2640行　instring コマンド処理ルーチンです．部分列とまったく同じ単語があった場合にはその単語から，なかった場合にはその部分列が入るべき位置をはさんでいる上の方の単語，すなわち，その部分列を含む最初の単語から表示します．部分列といっても頭から一致していなければなりませんから，INSTR 関数の値が１かどうか判断しています．１であれば，その部分列と単語の頭部が一致している単語はないことになりますので，〝not found〟を表示して，再び部分列を聞いてきます．2590行で，部分列を含む単語を表示します．2600行は，キーの入力待ちループです．もし，キーが押されれば，CR かどうか判断します．CR であれば，対応表の次の単語を表示し，CR でなければ，その単語の意味を表示します．

部分列による検索は，便利な機能です．通常の完全一致による検索は，長い名前を１文字まちがえたために，見つからなかったり，対応表には余分な文字が付いた形で登録されている場合には見つけることができません．ある項目について調べたいとか，関係のあるものを探し出す場合には，INSTR 関数を使えば，簡単に処理できます．

以上でこのプログラムの説明は終りです．このプログラムは，辞書といっても，INSTR コマンドがあるくらいで，ランダムアクセスプログラムの骨組みだけです．このプログラムを中心に，いろいろなランダムアクセスのアプリケーションプログラムを書くことができると思います．

```
1000 '
1010 ' dictionaly
1020 '
1030 CLEAR 5000
1040 DEFINT A-Z
1050 SIZE=500
1060 DIM W$(SIZE)'word array
1070 DIM R (SIZE)'record array
1080 DIM FS(SIZE)'free stack
1090 CR$=CHR$(13):LF$=CHR$(10)
1100 FLAG=0
1110 '
1120 ' read index
1130 '
1140 OPEN "dictio.idx" FOR INPUT AS #1
1150 N=0
1160 IF EOF(1) THEN 1210
1170 N=N+1
1180 R(N)=CVI(INPUT$(2,#1))
1190 INPUT #1,W$(N)
1200 GOTO 1160
1210 CLOSE #1
1220 '
1230 ' read free stack
1240 '
1250 OPEN "dictio.fre" FOR INPUT AS #1
```

```
1260 FR=0
1270 IF EOF(1) THEN 1310
1280 FR=FR+1
1290 FS(FR)=CVI(INPUT$(2,#1))
1300 GOTO 1270
1310 CLOSE #1
1320 '
1330 OPEN "dictio.dat" AS #1
1340 MAX=LOF(1)
1350 FIELD #1,255 AS MF$,1 AS L$
1360 '
1370 ' get command
1380 '
1390 PRINT 'line feed
1400 PRINT "command:";
1410 CM$=INPUT$(1)
1420 CM=INSTR("esaldi",CM$)
1430 IF CM=0 THEN BEEP:GOTO 1410
1440 ON CM GOSUB 1470,1720,1860,2190,2310,2490
1450 GOTO 1370
1460 '
1470 PRINT "exit"
1480 CLOSE #1
1490 IF NOT FLAG THEN 1690
1500 '
1510 ' write index
1520 '
1530 OPEN "dictio.idx" FOR OUTPUT AS #1
1540 FOR I=1 TO N
1550   PRINT #1,MKI$(R(I));W$(I);",";
1560 NEXT I
1570 CLOSE #1
1570 CLOSE #1
1580 '
1590 ' write free stack
1600 '
1610 IF FR=0 THEN 1690
1620 OPEN "dictio.fre" FOR OUTPUT AS #1
1630 FOR I=1 TO FR
1640   PRINT #1,MKI$(FS(I));
1650 NEXT I
1660 PRINT #1,CHR$(26);
1670 CLOSE #1
1680 '
1690 PRINT:PRINT "Good bye.":PRINT
1700 END
1710 '
1720 ' search
1730 '
1740 PRINT "search key word"
1750 PRINT
1760 KY$=""
1770 INPUT"search:";KY$
1780 IF KY$="" THEN PRINT "end of search":RETURN
1790 GOSUB 2110 'search
1800 PRINT 'line feed
1810 IF NOT FOUND THEN PRINT "not found":GOTO 1750
1820 GET #1,R(M)
1830 PRINT LEFT$(MF$,ASC(L$));
1840 GOTO 1750
1850 '
1860 ' append
1870 '
1880 PRINT "append new key word":PRINT
1890 KY$=""
```

```
1900 INPUT "append:";KY$
1910 IF KY$="" THEN PRINT "end of append":RETURN
1920 PRINT "meaning:"
1930 M$=""
1940 M1$=""
1950 LINEINPUT M1$
1960 IF M1$<>"" THEN M$=M$+M1$+CR$+LF$:GOTO 1940
1970 '
1980 GOSUB 2110
1990 IF FOUND THEN U=M:GOTO 2070
2000 FLAG=-1
2010 FOR I=N TO U STEP -1
2020   W$(I+1)=W$(I):R(I+1)=R(I)
2030 NEXT I
2040 N=N+1
2050 W$(U)=KY$
2060 IF FR THEN R(U)=FS(FR):FR=FR-1 ELSE MAX=MAX+1:R(U)=MAX
2070 LSET MF$=M$:LSET L$=CHR$(LEN(M$))
2080 PUT #1,R(U)
2090 GOTO 1890
2100 '
2110 ' binary search
2120 '
2130 L=0:U=N+1
2140 IF U-L=1 THEN FOUND=0:RETURN
2150 M=(U+L)¥2
2160 IF W$(M)=KY$ THEN FOUND=-1:RETURN
2170 IF W$(M)<KY$ THEN L=M ELSE U=M
2180 GOTO 2140
2190 '
2200 ' list up key words
2210 '
2220 PRINT "list up key words":PRINT
2230 IF N=0 THEN 2270
2240 FOR I=1 TO N
2250   PRINT W$(I),R(I)
2260 NEXT I
2270 PRINT
2280 PRINT "number of key words:";N
2290 PRINT "number of records:";MAX
2300 RETURN
2310 '
2320 ' delete key word
2330 '
2340 PRINT "delete key word"
2350 PRINT
2360 KY$=""
2370 INPUT"delete:";KY$
2380 IF KY$="" THEN PRINT "end of delete":RETURN
2390 GOSUB 2110
2400 IF NOT FOUND THEN PRINT "not found":GOTO 2350
2410 FLAG=-1
2420 N=N-1
2430 FR=FR+1:FS(FR)=R(M)
2440 FOR I=M TO N
2450   W$(I)=W$(I+1):R(I)=R(I+1)
2460 NEXT I
2470 GOTO 2350
2480 '
2490 ' instring search
2500 '
2510 PRINT "instring search"
2520 PRINT
2530 KY$=""
2540 INPUT "instring:";KY$
```

```
2550 IF KY$="" THEN PRINT "end of instring search":RETURN
2560 GOSUB 2110
2570 IF NOT FOUND THEN M=U
2580 IF NOT INSTR(W$(M),KY$)=1 THEN PRINT "not found":GOTO 2520
2590 PRINT W$(M)
2600 A$=INKEY$:IF A$="" THEN 2600
2610 IF A$=CR$ THEN M=M+1 :GOTO 2580
2620 GET #1,R(M)
2630 PRINT LEFT$(MF$,ASC(L$));
2640 GOTO 2520
```

4．5　「クレジット」←「ビジネスのために」

前に紹介した「辞書」で，ディスクからのデータの読み込み，データの書き込みの仕方は理解いただけたと思います．「クレジット」では，ディスクからのデータを読み込んで，そのデータを処理して再びディスクへ書き込む動作について述べます．このプログラムでは，3章で説明したレコードの分割利用も応用されています．

4．5．1　使い方

初めてのプログラムを走らせる場合には，2つのシーケンシャルファイルをあらかじめ，作っておく必要があります．この理由は，「辞書」と同じです．したがってまず次の命令列を実行します．

```
OPEN "credit.idx" FOR INPUT AS #1
CLOSE
OPEN "credit.fre" FOR INPUT AS #1
CLOSE
```

プログラムをRUNさせると，はじめに年月日を聞いています．これは，後で説明する請求書を発行するためのもので，80年5月1日でしたら，80／05／01と入力します．するとディスクを何度かアクセスしたのち，コマンド待ちとなり，「シゴト」と聞いてきます．コマンドには次の8種類あります．

1. オワリ
2. サーチ
3. ヘンコウ
4. イチラン
5. カイヤク
6. ツケ
7. シハライ
8. セイキュウ

それぞれのコマンドは頭の1文字のみ打ち込むことによって動作します．この動作は「辞書」と同じです．次に各コマンドの説明をしましょう．

オワリ　コマンド：このコマンドは，プログラムの終了コマンドです．対応表または請求日に変更があれば，書き込みを行います．「ゴクロウサン」と表示してプログラムを終了します．

サーチ　コマンド：このコマンドは，特定の一名の住所，先月までの残高，今月分の借金，請求日を表示します．「ナマエ？」と聞いてきますから，調べたい人の名前を入力します．対応表の中を探して，その人の名前を入力します．対応表の中を探して，その名前が見つかれば，その人のデータを表示して，再び「ナマエ？」と聞いてきます．もし見つからなければ，「ミツカリマセン」と表示して，再び「ナマエ？」と聞いてきます．コマンドを終了したい場合には，「ナマエ？」と聞いてきた時点で，キャリッジリターンのみを入力すれば，コマンドモードに戻って「シゴト」と聞いてきます．以下のコマンドも，コマンドモードへ，戻る方法は同じで

す．

ヘンコウ　コマンド：このコマンドは，すでに登録してある人のデータを変更するために使います．「ナマエ？」と聞いてきますから，変更したい人の名前を入力します．次に「ジュウショ？」と聞いてきますから，住所を変更したければ新しい住所を，変更したくなければキャリッジリターンのみを入力します．次々に「センゲツマデ　ノ　ツケ？」，「コンゲツブン　ノ　ツケ？」，「セイキュウビ？」と聞いてきますから，変更したならば新しいデータを，変更したくなければ，キャリッジリターンのみを入力します．全部聞き終ったところで，再び「ナマエ？」と聞いてきます．

イチラン　コマンド：このコマンドは，現在登録されているすべての人の，名前，住所，先月までの残高，今月分の借金，請求日を表示します．最後に登録されている人の総数，及びファイルの大きさをレコード単位で表示しています．このとき，人数とレコード数が違っていますが，これは，1レコードを4人で使っているためです．

カイヤク　コマンド：このコマンドは，客を表から削除するのに使います．ただし，まだ借金が残っているのに，解約しようとするとエラーとなります．

ツケ　コマンド：このコマンドは，付けで買い物をしたときに使います．「ナマエ？」と聞いてきますから名前を入力し，「キンガク？」と聞いてきますから金額を入力します．すでに登録されている客ならば今月分に加算します．初めての客ならば，住所と請求日を聞いてきますから入力します．

シハライ　コマンド：このコマンドは，借金が返済されたときに使います．「ナマエ？」と聞いてきますから，返済した客の名前を入力します．次に「キンガク？」と聞いてきますから，返済した金額を入力します．

セイキュウ　コマンド：すべての客の請求日を調べて，請求日に当っている客ならば，請求書を発行します．

4．5．2　プログラムの解説

このプログラムはリストを見ていただいてもわかるように，「辞書」を基本に発展させたものです．新たに3章で説明した，レコードの分割を応用していますし，MK シリーズ，CV シリーズによる数値データの変換と計算を理解できると思います．

1080行　数値配列D(　)は，客の請求日を記憶している配列です．

1130行～1150行　日付の入力です．キャリッジリターンのみ入力した場合には，入力変数の値に変化がないことを利用した，入力方法です．並通の変数ならば，入力変数並びの中に書いておくだけで上記の性質を利用できますが，DATE$であるとか，ランダムバッファ上の変数の場合は直接，入力変数並びに書くことはできません．したがって，もう1つ別に変数を用意して，その値を代入して，入力が終ってから再び元の変数に代入しなおすという方法を使います．この場合，D0$ がその役目を果します．

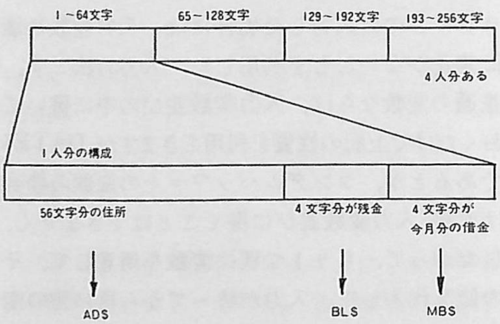
1160行～1270行　対応表の読み込みです．名前とレコード番号だけでなく，客の請求日も同じファイルに書いておいて，同時に読み込みます．請求日は255以下の整数なので ASC 関数を使って1バイトの数値データとして出し入れしてい

ます.(シーケンシャルファイルにおける CV シリーズと MK シリーズの利用の仕方は「辞書」に述べられています)

1390行で〝credit. dat〞を OPEN した後, FIELD 文で,変数への割付けを行っていませんが,これは後で述べるように,客のレコード番号に従って,その都度,割付けを行っているためです.

1410行～1500行　コマンドに従って各ルーチンに分枝する部分です.ここでも「辞書」と同じように,INSTR 関数による,コマンド調べを行っています.

1760行～1880行　サーチコマンドの処理ルーチンです.ヌルストリングを使って,キャリッジリターンのみの入力の検出,コマンドの終了判断を行っています.1860行で,対応するレコードを GET#しています.ここで,各レコードの構成を説明しましょう,各レコードは4人分になっています.



レコードの分割方法については,3章で説明したように,4人分の変数を用意して,4人別々のプログラムを作る方法と4つの要素を持つ配列を用意する方法と4人別々に FIELD 文を使ってその都度変数の割付けを行う方法の3つを紹介しました.ここでは,最後の FIELD 文をその都度作用させる方法を採用しました.対応表の中の配列R( )に入っている番号は,その客のデータが入っているレコード番号とは1対1に対応しているわけではありません,0,1,2,3はレコード番号1に,4,5,6,7はレコード番号2に,というように記録されています.客のレコード番号から,実際のレコード番号を算出し,その値に応じて,変数の割付けを行うのが,サブルーチン「レコードブンカツ」です.変数Rに客のレコード番号を入れて,サブルーチンへ飛ぶと,変数の割付けを行い,変数Rに実際のレコード番号を入れて戻ってきます.文字変数 AD$ は客の住所,文字変数 BL$ は前月までの差引残高,文字変数 MB$ は,今月分の借金です.

1890行～2100行　ヘンコウ　コマンド処理ルーチンです.変更なしの場合のために,変数 I, I$ を使っています.対応表といっしょに,請求日の配列を出し入れしていますので,請求日に変更があったか検出するために,変数D1を使っています.変更があった場合には,FLAG をセットしています.

2110行～2190行　2分検策サブルーチン.

2200行～2350行　イチランコマンド処理ルーチンです.対応表の頭から客のデータをディスクから読んでは,表示します.

2360行～2580行　カイヤクコマンド処理ルーチンです.2470行で,借金が0であるか検査しています.2570～2580行は,3章で説明した削除ルーチンに配列D( )の移動を加えたものです.

2590行～2980行　ツケコマンド処理ルーチンです.名前が登録してある場合には,2710行で,今月分の借金に,今入力された金額を加えてい

ます．登録されてない場合には，2790行～2830行で挿入を行っています，このルーチンは3章で説明したものに，配列D（ ）の移動を追加したものです．未使用レコードがスタックに積まれている場合は同じですが，積まれていない場合には，レコードが分割されているために余分な処理が必要になります．それが，2850行～2900行です．2850行で新しいレコード番号を割り当てます，1レコードで4人分ですから，残り3人分は未使用になるわけです，これを新しく，未使用スタックに積みます．スタックに一度に3つ積まれるのです．

2990行～3140行　シハライコマンド処理ルーチンです．3120行で残金の計算を行っています．
3150行～3500行　セイキュウコマンド処理ルーチンです．3220行で変数Dに日を代入しています．3210行～3230行のFOR-NEXTループで請求日に当っている客を探します．請求日に当っている客を見つけ出したなら，3260行からのサブルーチンに飛びます．変数BLに先月までの差引残高を，変数MBに今月の借金を代入します．借金の総計が負の場合は，請求書は発行しません．先月までの差引残高がある場合には利子が付きます，ここでは0.015にしておきました．3290行は，利子の四捨五入を行っています．

3510行～3560行　レコード分割割当のサブルーチンです．変数C%は，1レコード中，何番目の部分かを示します．文字変数DUMMY$は，位置をずらすための，ダミー変数です，変数としては用意されますが，文字領域は消費しません．

```
1000 '
1010 ' ｸﾚｼﾞｯﾄ
1020 '
1030 CLEAR 5000
1040 DEFINT R,D,F
1050 SIZE=500
1060 DIM NM$(SIZE)'ﾅﾏｴ
1070 DIM R  (SIZE)'ﾚｺｰﾄﾞ ﾊﾞﾝｺﾞｳ
1080 DIM D  (SIZE)'ｾｲｷｭｳﾋﾞ
1090 DIM FS (SIZE)'ﾐｼﾖｳ ﾚｺｰﾄﾞ
1100 CR$=CHR$(13):LF$=CHR$(10)
1110 FLAG=0
1120 '
1130 DO$=DATE$
1140 INPUT "ﾈﾝｶﾞｯﾋﾟ ｦ ﾆｭｳﾘｮｸｼﾃｸﾀﾞｻｲ";DO$
1150 DATE$=DO$
1160 '
1170 ' ﾀｲｵｳﾋｮｳ ﾖﾐｺﾐ
1180 '
1190 OPEN "credit.idx" FOR INPUT AS #1
1200 N=0
1210 IF EOF(1) THEN 1270
1220 N=N+1
1230 R(N)=CVI(INPUT$(2,#1))
1240 D(N)=ASC(INPUT$(1,#1))
1250 INPUT #1,NM$(N)
1260 GOTO 1210
1270 CLOSE #1
1280 '
1290 ' ﾐｼﾖｳ ﾚｺｰﾄﾞ ﾖﾐｺﾐ
1300 '
```

```
1310 OPEN "credit.fre" FOR INPUT AS #1
1320 FR=0
1330 IF EOF(1) THEN 1370
1340 FR=FR+1
1350 FS(FR)=CVI(INPUT$(2,#1))
1360 GOTO 1330
1370 CLOSE #1
1380 '
1390 OPEN "credit.dat" AS #1
1400 MAX=LOF(1)
1410 '
1420 ' ｼｺﾞﾄ ｳｹﾂｹ
1430 '
1440 PRINT
1450 PRINT "ｼｺﾞﾄ ";
1460 CM$=INPUT$(1)
1470 CM=INSTR("ｵｻﾍｲｶﾂｼｾ",CM$)
1480 IF CM=0 THEN BEEP:GOTO 1460
1490 ON CM GOSUB 1520,1770,1900,2200,2360,2600,3000,3160
1500 GOTO 1420
1510 '
1520 PRINT "ｵﾜﾘ"
1530 CLOSE #1
1540 IF NOT FLAG THEN 1740
1550 '
1560 ' ﾀｲｵｳﾋｮｳ ｶｷｺﾐ
1570 '
1580 OPEN "credit.idx" FOR OUTPUT AS #1
1590 FOR I=1 TO N
1600   PRINT #1,MKI$(R(I));CHR$(D(I));NM$(I);",";
1610 NEXT I
1620 CLOSE #1
1630 '
1640 ' ﾐｼﾖｳ ﾚｺｰﾄﾞ ｶｷｺﾐ
1650 '
1660 IF FR=0 THEN 1740
1670 OPEN "credit.fre" FOR OUTPUT AS #1
1680 FOR I=1 TO FR
1690   PRINT #1,MKI$(FS(I));
1700 NEXT I
1710 PRINT #1,CHR$(26);
1720 CLOSE #1
1730 '
1740 PRINT:PRINT "ｺﾞｸﾛｳｻﾝﾃﾞｼﾀ":PRINT
1750 END
1760 '
1770 ' ｻｰﾁ
1780 '
1790 PRINT "ｻｰﾁ"
1800 PRINT
1810 KY$=""
1820 INPUT "ﾅﾏｴ ";KY$
1830 IF KY$="" THEN PRINT "ｻｰﾁ ｵﾜﾘ":RETURN
1840 GOSUB 2120
1850 IF NOT FOUND THEN PRINT "ﾐﾂｶﾘ ﾏｾﾝ":RETURN
1860 R=R(M):GOSUB 3520:GET #1,R
1870 PRINT AD$;CVS(BL$);CVS(MB$);D(M)
1880 GOTO 1800
1890 '
1900 ' ﾍﾝｺｳ
1910 '
1920 PRINT "ﾍﾝｺｳ"
1930 PRINT
1940 KY$=""
1950 INPUT "ﾅﾏｴ";KY$
```

```
1960 IF KY$="" THEN PRINT "ﾍﾝｺｳ ｵﾜﾘ":RETURN
1970 GOSUB 2120
1980 IF NOT FOUND THEN PRINT "ｱﾘﾏｾﾝ":GOTO 1930
1990 R=R(M):GOSUB 3520:GET #1,R
2000 I$=AD$
2010 INPUT "ｼﾞｭｳｼｮ        ";I$:LSET AD$=I$
2020 I=CVS(BL$)
2030 INPUT "ｾﾝｹﾞﾂ ﾏﾃﾞ ﾉ ﾂｹ";I:LSET BL$=MKS$(I)
2040 I=CVS(MB$)
2050 INPUT "ｺﾝｹﾞﾂﾌﾞﾝ ﾉ ﾂｹ";I:LSET MB$=MKS$(I)
2060 D1=D(M)
2070 INPUT "ｾｲｷｭｳﾋﾞ       ";D(M)
2080 IF NOT D1=D(M) THEN FLAG=-1
2090 PUT #1,R
2100 GOTO 1930
2110 '
2120 ' ﾆﾌﾞﾝｹﾝｻｸ
2130 '
2140 L=0:U=N+1
2150 IF U-L=1 THEN FOUND=0:RETURN
2160 M=(U+L)¥2
2170 IF NM$(M)=KY$ THEN FOUND=-1:RETURN
2180 IF NM$(M)<KY$ THEN L=M ELSE U=M
2190 GOTO 2150
2200 '
2210 ' ｲﾁﾗﾝ
2220 '
2230 PRINT "ｲﾁﾗﾝ":PRINT
2240 IF N=0 THEN 2320
2250 '
2260 FOR I=1 TO N
2270   R=R(I):GOSUB 3520:GET #1,R
2280   PRINT NM$(I),AD$
2290   PRINT TAB(15);CVS(BL$),CVS(MB$),D(I)
2300   PRINT
2310 NEXT I
2320 PRINT
2330 PRINT "ｷｬｸ  ﾉ ｶｽﾞ  ";N;"ﾆﾝ"
2340 PRINT "ﾌｧｲﾙ ﾉ ｵｵｷｻ";MAX;"ﾚｺｰﾄﾞ"
2350 RETURN
2360 '
2370 ' ｶｲﾔｸ
2380 '
2390 PRINT "ｶｲﾔｸ"
2400 PRINT
2410 KY$=""
2420 INPUT "ﾅﾏｴ ";KY$
2430 IF KY$="" THEN PRINT "ｶｲﾔｸ ｵﾜﾘ":RETURN
2440 GOSUB 2120
2450 IF NOT FOUND THEN PRINT "ﾐﾂｶﾘﾏｾﾝ":GOTO 2400
2460 R=R(M):GOSUB 3520:GET #1,R
2470 IF CVS(BL$)+CVS(MB$)=0 THEN 2520
2480 BEEP
2490 PRINT USING "ﾏﾀﾞ ｻﾞﾝｷﾝ ｶﾞ ¥########, ｱﾘﾏｽﾖ｡";CVS(BL$)
2500 GOTO 2400
2510 '
2520 FLAG=-1
2530 N=N-1
2540 FR=FR+1:FS(FR)=R(M)
2550 FOR I=M TO N
2560   NM$(I)=NM$(I+1):R(I)=R(I+1):D(I)=D(I+1)
2570 NEXT I
2580 GOTO 2400
2590 '
2600 ' ﾂｹ
```

```
2610 '
2620 PRINT "ツケ"
2630 PRINT
2640 KY$=""
2650 INPUT "ナマエ ";KY$
2660 IF KY$="" THEN PRINT "ツケ オワリ":RETURN
2670 GOSUB 2120
2680 INPUT "キンガク";CH
2690 IF NOT FOUND THEN 2770
2700 R=R(M):GOSUB 3510:GET #1,R
2710 LSET MB$=MKS$(CVS(MB$)+CH)
2720 PUT #1,R
2730 GOTO 2500
2740 '
2750 ' シンイリ
2760 '
2770 PRINT "シンイリ"
2780 FLAG=-1
2790 FOR I=N TO U STEP -1
2800   NM$(I+1)=NM$(I):R(I+1)=R(I):D(I+1)=D(I)
2810 NEXT I
2820 N=N+1
2830 NM$(U)=KY$
2840 IF NOT FR=0 THEN R=FS(FR):FR=FR-1:R(U)=R:GOSUB 3520:GET #1,R:GOTO 2930
2850 R=MAX*4
2860 FOR I=1 TO 3
2870   FS(FR+I)=R+I
2880 NEXT I
2890 FR=FR+3
2900 MAX=MAX+1
2910 R(U)=R
2920 GOSUB 3520
2930 INPUT "ジュウショ  ";I$:LSET AD$=I$
2940 INPUT "セイキュウビ ";D(U)
2950 LSET BL$=MKS$(0)
2960 LSET MB$=MKS$(CH)
2970 PUT #1,R
2980 GOTO 2630
2990 '
3000 ' シハライ
3010 '
3020 PRINT "シハライ"
3030 PRINT
3040 KY$=""
3050 INPUT"ナマエ";KY$
3060 IF KY$="" THEN PRINT "シハライ オワリ":RETURN
3070 GOSUB 2120
3080 PRINT
3090 IF NOT FOUND THEN PRINT "イマセン":GOTO 3030
3100 INPUT "キンガク ";I
3110 R=R(M):GOSUB 3520:GET #1,R
3120 LSET BL$=MKS$(CVS(BL$)-I)
3130 PUT #1,R
3140 GOTO 3030
3150 '
3160 ' セイキュウ
3170 '
3180 PRINT "セイキュウ"
3190 PRINT
3200 D=VAL(RIGHT$(DATE$,2))
3210 FOR I=1 TO N
3220   IF D(I)=D THEN GOSUB 3260
3230 NEXT I
3240 RETURN
3250 '
```

```
3260 R=R(I):GOSUB 3520:GET #1,R
3270 BL=CVS(BL$):MB=CVS(MB$)
3280 IF BL+MB<=0 THEN IN=0:GOTO 3480
3290 IN=INT(BL*.015+.5)
3300 IF IN<0 THEN IN=0
3310 PRINT STRING$(79,"-")
3320 PRINT TAB(20);"***ｾｲｷｭｳｼｮ***";TAB(50);DATE$
3330 PRINT
3340 PRINT NM$(I),"ｻﾏ"
3350 PRINT TAB(16);AD$
3360 PRINT
3370 PRINT TAB(30):PRINT USING "ｺﾝｹﾞﾂ ﾌﾞﾝﾉ ﾂｹ ¥########,";MB
3380 PRINT TAB(30):PRINT USING "ｾﾝｹﾞﾂ ﾏﾃﾞﾉ ﾂｹ ¥########,";BL
3390 PRINT TAB(30):PRINT USING "ﾘｼ              ¥########,";IN
3400 PRINT
3410 PRINT TAB(30):PRINT USING "ｺﾞｳｹｲ           ¥########,";BL+MB+IN
3420 PRINT
3430 PRINT TAB(16);"ｼﾞｮｳｷ ｾｲｷｭｳ ﾓｳｼｱｹﾞﾏｽ"
3440 PRINT
3450 PRINT TAB(30);"ｱｽｷｰ ｼｭﾊﾟﾝ K.K."
3460 PRINT TAB(30);"T110 ﾄｳｷｮｳ ﾄ ﾐﾅﾄ ｸ ﾐﾅﾐｱｵﾔﾏ 15-16-1 ｱｵﾔﾏﾋﾞﾙ 5F"
3470 PRINT STRING$(79,"-")
3480 LSET BL$=MKS$(BL+MB+IN):LSET MB$=MKS$(0)
3490 PUT #1,R
3500 RETURN
3510 '
3520 ' ﾚｺｰﾄﾞ ﾌﾞﾝｶﾂ
3530 '
3540 C%=(R MOD 4)*64:R=R¥4+1
3550 FIELD #1, C% AS DUMMY$, 56 AS AD$, 4 AS BL$, 4 AS MB$
3560 RETURN
```

5 Q&A

5章では，Disk BASICの著者それぞれに，Disk BASICの機能やディスクに対する疑問を読者の方に明確に理解して頂けるように，QUESTIONをしてみました．
初歩的なことから，ディスクの高度な部分にまで質問するように心掛けて行ないました．
他の章で，理解できなかった部分は，この5章で補って頂けると思います
ディスクのハード，ソフト面について色々な角度から，多種多様の質問を行ないました．
解答者は，Disk BASICの著者大野氏，飯田氏，鈴木氏です．
質問を担当したのは，出版部の田口です．
文章中の敬称は省かせて頂きます．

1 ディスクとは何か

田口：コンピュータの拡張インターフェイスとして，ディスクと呼ばれているものがありますが，そもそも，何が目的で何をさせるものなのでしょうか．

鈴木：そもそもの働きをいえば当然の事として，データを記憶保存しておくことです．テープレコーダみたいなもので，言ってしまえばテープレコーダは人間の声や音楽を記憶するもの．それに対して，ディスクはコンピュータの，たとえばPCにつながったディスクになら当然PCからのデータを記憶するものだと解釈できます．

田口：プログラムや，データの保存をしておくものですね．

鈴木：プログラムも広い意味では1つのデータですからね．

田口：ディスクをPCにつけた場合のメリット，デメリットはなんですか．

鈴木：この本の第0章にも記述していますがディスクのメリットとして，データの入出力が高速である，容量が大きい，などがあげられます．

カセットでデータを処理するプログラムを書いてみるとそのメリットが，良く理解できると思いますよ．例えば，PC#1に載っていた名簿管理プログラムが，データ処理の良い例です．このプログラムはデータファイルをカセット上に作っているのですが，ファイルの読み書きに時間がかかるという欠点がありました．それと，データファイルを読むのに時間がかかるということは，読んでいる最中に，ERROR が出る事が多くなりますね．

こういった欠点をカバーできる記憶装置として Disk があるわけです．

Disk の速度と信頼性はカセットの比ではないわけです．もっとも，高速で信頼性が高いといっても限度はあります．ディスクのディメリットはそのあたりから生じてくるわけです．

デメリットは，全く逆の事がいえると思います．

データ処理が遅い，容量が小さいなどです．

これは用途によるのですが，大量のデータを高速に処理しなければならない用途があるわけです．要はディスクがその用途のどのあたりまでをカバーできるかが問題なわけです．ビジネス等の分野や，科学技術計算の分野ではカバーできなくなることもあるということですね．

この事は，第１章で詳しく説明してあります．

2　ディスクの働き

田口：ディスクではディスケットというものが使われますが，ディスケットはどういう働きをするものなのですか．

鈴木：ディスケットの働きを理解するには，ドライブとディスケットの役割りを比較してみることです．これは丁度テープレコーダとカセットテープの役割りと考えられるわけです．

田口：テープレコーダがディスクで，カセットテープがディスケットになるわけですか．

鈴木：その通りです．実際に記憶を担当するものが，ディスケットであり，ディスケットに記憶させるためのものがドライブやインターフェイスになるわけです．

田口：ディスケットにデータを落とすとか，記憶させるというふうに言いますが，どのような形でディスケットにデータが記憶されているのでしょうか．

飯田：ディスケットは，磁気テープを１枚の円盤にしたようなものです．その上に，テープレコーダーで録音するように，磁性的にデータを記録します．そして，レコードの様に，書き込み，読み出し兼用のヘッドが接触しながら，ディスケットが回転して，データを読み書きするわけです．通常コンピュータのメモリ上には，基本的に８ビットでパラレルに入っているわけですけれども，これをシリアルに変換して記憶しています．

3　トラックとセクタ

田口：シリアルやトラック，セクタといった言葉の意味を知らない人がいると思います．ディスケットの中でセクタ，トラックはどのようにして分けられているのですか．又，セクタ，トラックとはどのように管理されているのですか．

飯田：トラックは，ヘッドの通る道筋のことです．モノラルのテープレコーダなら行きと帰りの２トラック，ステレオテープレコーダなら行きの２本，帰りに２本で計４本のトラックがあります．ディスケッ

トにはトラックが何と35本，しかもエンドレスになっているわけです．

セクタは，1本のトラックを扱い易いように，16個に分けたものです．ヘッドをトラック数だけ用意しておくのが勿論，一番良いわけですが，フロッピーの場合は一個のヘッドを用意してそれを，ステッピングモーターと呼ばれるモータで移動するわけです．そして目的のトラックまでたどり着くと次に目的のセクタをどうやって呼び出すか，という事なのですが，まずデータはシリアルに入っているので，1と0の羅列なんです．ですからどこから必要なバイトが始まって，どこで必要なバイトが終るかというのは，全然わからないのです．そこで，それをある程度区切ればいいのですが，例えばトラックを1セクタにすると，それを同期するチャンスは1回になります．同期するというのは，シリアルとパラレルを変換する最初のタイミングを握むことです．それでは，1周につき，1回しかチャンスがないので，1回失敗するともう1回まわってくるまでそのチャンスは来ないのです．ですから，時間的なロスが大きくなるので，トラックの情報を適当に分けてやり，それによってまず必要なタイミングを得る為のチャンスを作るのです．例えばカセットテープで256バイトおきにマークを入れたりするというのと似たような事なのです．つまり同期のチャンスを作るというような意味です．

大野：他に，1トラックにセクタが1つしかないとすると，1トラック中の或データだけが欲しい場合，最初から読んでいかなければなりませんが，1トラックをセクタに分けることによって，必要なデータをすぐに取り出すことができます．

田口：トラックの中に1セクタなんて事は，通常ありませんが，その場合は，1つ1つ読んでいかなければならないという事ですね．

一同：そうです．

田口：それからもう1つ，用語事典のようになってしまいますが，シリアルとかパラレルについて，お聞きしたいのですが．

大野：シリアルとは直列，パラレルとは並列という意味です．コンピュータの内部では，データはパラレルで扱われますが，ディスクにはシリアルで記憶されます．

飯田：何故シリアルで入れるのかというと，別にヘッドが8個用意されていればパラレルで入れるのが一番良いのだけれど，高価になります．シリアルで入れれば，ヘッドが1つで済みます．

4　MOUNTの働き

田口：まずディスクの中にディスケットを差し込み，必ず Disk BASIC のシステムを呼び出した次の動作に MOUNT，という命令がありますが，何故 MOUNT をするのか，その必要性とは一体何なのでしょうか．

大野：少し話が飛びますが，ディスケット内に保存されている情報を扱う場合に，ユーザーはファイル名という名前によって論理的なものとしてファイルをとらえて，それに対してアクセスしようとします．ところが，実際にディスケットを扱うためには，その名前のファイルがディスケット内のどの場所にあるかがわかっていなければなりません．ユーザーの側としては，ファイルを完全に名前だけで参照したいのですから，ファイルがどこにあるかという情報は BASIC の方で知っていなければならないことになります．デ

ィスケット上での記憶はセクタという単位で扱かわれますが，1セクタには256バイトしか記憶できません．そのために，ファイルの大きさが何Kバイトにもなり，1つのセクタに入りきらない場合には，1つのファイルの内容が複数のセクタ,あるいは複数のトラックにまたがって記憶されることになるわけです．ですからBASICはファイルの内容がディスク内のどこのトラック，セクタから始まって，どことどこのセクタにしまわれているかということを知っていなければならないことになります．

このように，BASICはファイルの名前とそのファイルの使用しているディスケットの領域との対応関係を管理する役割りを持っています．この対応関係はそのディスケットに固有なものですから，ファイルの内容とともに同一のディスケット上に記録しておきます．MOUNTは，この対応関係をディスケットから読み出し，メモリ中にロードする働きを持っています．ファイルを参照する際には，この対応関係が常に必要となりますから，MOUNTによりメモリ中に読み込んでおけば，その後はメモリ中の対応関係を用いて高速に処理ができるようになります．

田口：それでDisk BASIC中のMOUNT命令が必ずいるという事なのですね．
それからもうひとつ．
I/Oポートが外に出ている必要性及びメリット，デメリット的なものとは一体何でしょうか．

飯田：何故I/Oポートが外に出ているのかという事について云えば，それは基本的にはまず拡張ユニットを前提にしていたからです．インテリジェント化されているとは，ドライブ自身1つのコンピュータになっているということです．通常フロッピードライブ本体を制御するためには，複雑なプロセスが必要です．ヘッドを移動して，ヘッドをロードして，読みたいセクタを探して，など，など．これらをPC本体にやらせるのは，大変です．したがって，もう1台ディスクドライブする為の専用のCPUを用意し，それら全てをやらせて，PC本体からは，何トラックの何セクタを読め，というような高度な命令を行うだけで済むようしています．

田口：そこで，I/Oポートの信頼性が一番問題になると思いますが，ディスクの信頼性は勿論，I/Oポートの信頼性がいい加減なようでは，ディスクは不能になってしまいます．I/Oポートの信頼性はどの位なんでしょうか．

飯田：信頼性というより寧ろ，安全性はあります．何故かというと，I/Oポートでは，ホストとドライブに区切られているので，ドライブに事故が起ってもホストまでは影響はないし，ホストに事故が起ってもドライブにまで及ぶ事はないのです．

5　ディスクの起動手順

田口：ディスクは何故リセットスイッチで動き出すようになっているのですか．他の機種ではソフトウェアでやっているようですが．

飯田：ハード的には必然性はありません．

大野：リセットスイッチを入れなくてもパワーを入れた時点でディスクが動くことはあります．

田口：ハード的には必然性がないという事はソフト的には，関係があるのですか．

大野：ソフトウェア的には，リセットスイッチによって，ディスクがブートする必然性は当然あります．ソフトの方で管理して

いるので，リセットした時点でディスクがつながっていなければ普通のN-BASICとしてスタートします．リセットした時点でN-BASICがPC本体にディスクドライブがつながっているかどうかをチェックするわけです．つながっていれば，ドライブの1にシステムディスケットが入っているという事を前提としてそこからDisk BASICを読み出します．以上の作業をするというシーケンスがプログラムで組まれているのです．

6　ディスクの性能

田口：ディスクの性能に記憶容量がありますが，それが大きいのか小さいのか，その辺をもう少し詳しくお願いします．

鈴木：計算によれば，月刊アスキーの紙面に字をびっしりうめて，32～36ページ位が1枚のディスケットに（システムディスクではなくて，完全に空のディスク）に入りますから，結構量が大きいのでビックリしています．

大野：倍密度ですから普通のミニディスケットの倍入るということです．

田口：それから，ディスクをPCにつけた場合はどれだけ便利になるのでしょうか．

鈴木：ディスクをつけて一番便利になるというのはカセットの時と比較するとよくわかります．例えば，カセットしか使わなかった時には，カセットはほとんどプログラムのLOAD, SAVE(PCではCLOAD, CSAVE)に使われています．しかしディスクの場合，当然それも出来るのですが，プログラム中でデータを処理する能力が，カセットよりもずっと強力になるわけです．たとえば，名簿管理プログラムの名簿のデータの処理等ですね．

7　ディスク上のシステムソフト

田口：どうしてディスクにシステムソフトが入っているのですか．

飯田：ドライブを持っていないユーザーの為を考えているのです．ドライブを動かす為には，ある程度の量のソフトがいるのですが，ドライブを持っていないユーザーには，PCにそれだけのソフトを積むとその分だけメモリを使えなくなってしまうのです．それで，ドライブがついていない時にはその不用な部分をユーザーがプログラムに使えるようにしようというのが，目的です．

8　REMOVEの働き

田口：PC本体のメモリの節約の為にシステム・ソフトをディスク中に用意しているという事ですね．
システム・ソフトはメモリ内におさまり，次の順序としてMOUNTするのが第一の条件です．その後はどうすればディスケット中のデータを読み出す事が出来ますか．

鈴木：データを読み出す場合も，書込む場合も，扱うデータの型によって操作を変えなければなりません．プログラムファイルを読む場合と，データファイルを読む場合とでは操作が変わってくるわけです．さらにデータファイルには，シーケンシャルファイルとランダムファイルがありますから，このどちらかによっても操作を変える必要がありますね．
具体的に書けば，プログラムファイルの読出しにはload，書込みには，saveランダムファイルの読出しにはGET#，書込みにはPUT#，シーケンシャルファイ

ルの読出しには INPUT#，書込みには PRINT#を使用することになります．

読出し，書込み用以外のステートメントも，対象にとするファイルの型が限られています．2章ではステートメントや関数を，対象とするファイルの型によって分類して説明しているわけです．

田口：ディスケットの使用が終って，ドライブからディスケットを抜き出す前に必ず remove するのが大事なのですが，remove の持っている役割とは何なのでしょうか．

大野： mount というのは PC にディスケットを「はい，入れましたよ」というのであり，「もう，ディスケットを出しますよ」というのが remove なわけです．勝手に人間がディスケットを差し返えてもそれはもう PC には分らないわけです．ところがディスケットには，入っている情報を管理する為に必要な情報というのがあるわけです．ディスケットに対して実際に書き込みを行った場合，つまりディスケットの持っている情報が増えた場合は，新しいファイルを作って，そこに情報が増えたということを入れて，そのファイルがディスクのどの部分から始まっているという様な情報をディスク上にちゃんと書いておく必要があるのです．

remove をするというのは，ディスケット上の書き込みやその他の行為がすべて終ったので，ドライブから抜き取るという事を PCに知らせる役目も持っています．

田口：もし remove をせずにドライブからディスケットを抜くとか， remove を忘れてディスク本体の電源を切ってしまった場合，ディスケットが受ける影響はどんなものでしょうか．

飯田：remove を忘れたからといって，ディスクがダメになるとはかぎりません．mount してから読み出しのみ行った場合には何の影響も与えませんが，書き込みや削除を行った場合にはディスクをダメにしてしまうことがあるのです．すなわち，FAT に変更があった場合には必ず remove をしなければなりません．remove を忘れた時にデータが壊れるわけではありません，次回，マウントした時に読み出される FAT は昔のままですから，そこでディスクに書き込みを行うと，大切なファイルを壊してしまうことがあるのです．

田口：つまりディスケットの情報は大切なものなので， remove を忘れてはならないという事ですね．

一同：そうです． remove の大切さを忘れないで欲しいですね．

9 セーブの種類

田口：次に大切なものとして，SAVE コマンドがありますが，これには通常の SAVE の他にアスキー SAVE というのがありますが，通常の SAVE とはどこが違うのでしょうか．

鈴木：プログラムを SAVE して，プログラムファイルを作る為のコマンドなんです．これは BASIC の機能になりますが，N-BASIC でも Disk BASIC でも一緒で，一種のコンパイラみたいな働きをもっていて，人間がキーボードからプログラムを入力し，それをメモリー中に納める時には，もうプログラムはコンピュータ用に形が変わっているのです．コンピュータ用にプログラムの形を変えるという事はメモリが少量で済み，ある程度実行も早くなるという事です．

このコンピュータ用に形を変えたプログラムを内部形式のプログラムと呼びます．それに対して，人間がキーボードから入力した形のプログラムをアスキー形式のプログラムと呼びます．普通の save の場合には，ディスクに内部形式の形で記録するわけですよ．アスキー SAVE というのはアスキー形式のプログラムをディスクに記録するのです．内部形式のプログラムの方がメモリが少量で済みますから，save するのに必要な時間も短いことになります．カセットへ記録する場合は，そんな理由から内部形式のプログラムを記録するわけですが，ディスクの場合には，速度が速いので，内部形式で記録しようと，アスキー形式で記録しようと，記録に要する時間はそんなに変わりませんから，どちらの形で記録しようとかまわないわけです．

大野：アスキー SAVE では，LIST を取った時に出てくるのと全く同じ内容がディスクに記録されているのです．

鈴木：ですから，たいした違いが出ないかもしれませんが，長いプログラムを SAVE すると長さが変わってくる事があります．

10　DATAファイルの種類

田口：次にファイルの事ですが，シーケンシャルファイルとランダムファイルの違いは何であり，ユーザーが使う場合にはどちらが便利なのでしょうか．

鈴木：まず違いについてですが，これはカセットを例にとって説明します．

シーケンシャルファイルというのは，データが非常に長い１本の線の様に続いているものです．データを読み出す時には，最初から順々にデータを読み出して行き，いつでも読めるのは常に自分の手元に来た情報だけしかないわけです．次を読むとか，その奥にあるデータを読む為には，根気よく１つ１つ追ってゆくしかないというのがシーケンシャルファイルです．

これに対し，ランダムファイルとは，たとえばカセットテープに対するレコードのようなものです．カセットが一次元上であるのに対し，レコードは面上に置いてあります．つまり二次元上にデータがあるわけですから．場所さえ指定すれば，カセットの様に最初から聞いて行かなくても，データを持ってこられるのです．

これがランダムファイルです．

田口：シーケンシャルファイルというのは，ベルトコンベアにのってくるデータを自分の欲しいデータが来るまでまってから取り出す，という事ですね．そしてランダムファイルの方は，いつでも自分の欲しいデータを取ってくる事が出来るわけですね．

鈴木：どちらが便利かというと，それは非常にむずかしい所なのですが，当然，データがどこからでも取り出せるという意味では，ランダムファイルの方が非常に便利な様に思えます．しかし，それなりの手順というものがありますから，ランダムファイルの場合には，どこを持ってくるかという適確な意思表示をしなければならないわけです．こちらで場所を管理するという事です．

大野：要するに欲しい情報がどこにあるかという事を知っていなければならないわけですね．

鈴木：それに対して，シーケンシャルファイルは非常に単純なわけです．たえず，もってこられるのは１個だけなのですから，欲しいデータが出てくるまで待ってさえいればよい．つまり不便な様でいて使い

易いとも云えるわけです．ですから実際にどちらを使うかというのは，非常にむずかしい所なのです．ランダムファイルというのは，シーケンシャルファイルの機能も持ちますから，ランダムファイルを使って，シーケンシャルファイルの様な動作をやらせても悪くないのです．それなりに管理をしっかり，つまり最初から読んで来るというようにすれば良いのです．

田口：すると何故，ランダムファイルとシーケンシャルファイルをわけ，それぞれの機能を持たせなければならないのかという事になりますね．

鈴木：ランダムファイルというのは使い方が圧倒的に面倒なんですよ．

飯田：世の中の仕事の殆どはシーケンシャルファイルで間に合うのですが，より高度な機能を持ったファイルを作りたいならば，ランダムファイルにしなければならないという事です．

たとえば１日の仕事をまとめて処理するなんて事が多い訳ですよね．いつも大きなファイルの書き換えをいつもやっている訳ではないでしょう．

大野：ランダムの方が情報処理が早いのです．どこのデータでも持ってこられますからね．

飯田：あとシーケンシャルファイルというのは，バックが出来ないというのが致命的です．一度データを読んだら，次に読むときは，もう一度始めから検索のやり直しということになって，そこが，不便なところですね．

11　OPENとCLOSEの働き

田口：ファイルについてはこの辺にしまして，ディスクのOPEN，CLOSEの機能について，知りたいのですが．

鈴木：具体的に OPEN とCLOSE の機能をファイルとファイル番号とを対応づけることで説明いたしましょう．

プログラム中でデータファイルを扱う時なんですが，このとき，ファイルの名前では扱わずに番号で扱うのです．すると最初にファイル番号何番目はどのファイル名にあたるかを対応づけしなければならなくなります．OPEN はこれを宣言するのだとみているわけです．

非常におもしろいのは，対応する動作で，ファイルの番号というのは，最大で15まで使えるわけですから，どのファイルがどのファイル番号に対応するかを OPEN で指定してプログラム中で使うわけです．

ディスケット上のファイル群のうち最大15個までファイル番号と対応づける事によって使える訳です．が，思い出してもらいたいのは，カセットです．

カセットというのは，シーケンシャルファイルを扱いますが，同時に，ファイル１個しか，使えないのです．

PRINT#-1といったコマンドがカセットにデータファイルを作るコマンドなんですが，カセットでは，たった１個しかオープンできないファイルのファイル番号が，－１に固定されているわけです．

ところが，ディスク上では，ファイルが一杯あって同時に15個まで使える様になっていて，番号も15まで使ます．そこで，対応づけをしなければならないわけです．それでわざわざ最初に宣言して使うのです．

その対応づけを解くのが CLOSE なのです.
CLOSE したファイル番号は,又他のファイルに対応づけして使う事ができるわけです.

大野：ディスク上にはファイルが沢山あってディスクというのは，一種のカセットのライブラリーの様な役目を果していて，ファイルを OPEN するという事は，ライブラリーの中から自分の欲しいカセットをもってくる事ということです.
そして CLOSE というのはライブラリーの中にカセットを戻す事だと考えることができます.

12 ディスクの利用範囲

田口：PC にディスクをつけて拡張した人やこれから，ディスクをつけて拡張したいと考えている人のために，ディスクの利用範囲はどの程度なのでしょうか.

飯田：信頼性においては，あまり高いとはいえませんが，ディスクを正しく扱えるようになった人には，それ相当の使い方ができると思います.
情報処理が早い，容量が大きいなどのカセットインターフェイスを使っているときとは，くらべものにならないほどデータを処理する範囲が広くなります.
ディスクだけでビジネスに使おうと思う人は，ディスケットのバックアップが絶体に必要です.
remove をわすれたり,その他の悪条件が重なってディスケットのデータが壊れることがあります.

田口：ディスクの耐久性はどのくらいなのでしょうか.

大野：マニュアルには，特に時間的寿命は記載されておりませんが,まあ，設計寿命は，一応5年と私たちは考えています.
使用目的，用途，環境によって，多少使用可能年数が変わるでしょう.

飯田：機械部で寿命は決まりますから，部分的な交換で，寿命が伸びることは充分考えられることです.

13 ディレクトリとは何か

田口：ディスケットの話になりますが，ディレクトリについても知りたいのですが.

大野：先程から話しているようにファイルがディスケットの上のどこにあるか，ディスケット中に一体どういう種類のファイルがあるのかという情報もディスケット上においておかなければならないのです.
そういう情報を入れておく場所がディレクトリであり，FAT であるわけです.
ディレクトリは台帳という意味なんですけれど，ファイルの名前と住所が書いてあるものだと思って頂ければいい訳です.
つまり，ファイルの名前とそのファイルのしまわれているディスケット上の位置を保持しているのがディレクトリです.
実際には，ファイルの大きさに応じてファイルの占める領域も大きくなるので，ディレクトリには，ファイルの先頭の部分が使用しているセクタの位置が保持されていて，続くファイルの内容がどこにしまわれているかは FAT に記録されています.
このようにして，ディレクトリはファイルの名前と，ファイルの内容が保持されている位置とを対応づけるという役割りを果たしているわけです.

14　モジュール化の方法

田口：ディスクによって，少々長いプログラムでも扱えるようになった，としても，あまり長いものやちょっと復雑なプログラムはうまくいかない場合があると思いますが，その時，プログラムのチェインという言葉がありますが，具体的にチェインとはどういった機能なのでしょうか．

鈴木：２つ以上のプログラムを連続して走らせる方法として，二つの方法があるわけです．
１つは２つのプログラムを最初に動かす前にMERGEで１つのプログラムに設定してしまうのです．
MERGEのところを読んでもらえば良くわかると思います．
MERGEを使えば，当然１つのプログラムになりますから，それでOKです．もう１つの方法は，これは最後の手段なんですが，メモリに入りきらなかった場合，（１本のプログラムが長すぎて）プログラムを２本に分けるわけです．

田口：今度は１本のプログラムを２本に分けるわけですね．

鈴木：１本のプログラムを２本に分けて，片方を動かして処理が終わったら，その時点でディスクから残りのプログラムをロードして，ロードされたプログラムに実行を移すという方法があるわけです．
その時に，例えばファイルを使ったとします．
２番目のプログラムをロードするとき，通常のRUNとするとファイルがCLOSEされてしまいます．それでは困るので，OPENされているファイルをCLOSEしないで，ファイルはそのまま残して動かそうという方法があるのです．
この方法が，RUN〝ファイル名〃，Rと，LOAD〝ファイル名〃，Rです．
この方法はディスクがいかに速いとはいえ大変時間のかかるわけで，やらないほうが無難でしょう．
run〝ファイル名〃，rとload〝ファイル名〃，rは機能は全く同じで，runはロードしてあとのプログラムにrunをかける動作なのです．
load & runをOne Stepでやってくれるのです．
しかし，一方のプログラムから他のプログラムをサブルーチンのようにして呼び出す事は出来ないのです．

田口：サブルーチンとして呼び出すことはなぜできないのですか．

鈴木：まったく方法がないわけではないのですが，大変難かしいことですね．
又，プログラムをOPENしたまま，runやloadをすると変数類はすべてクリヤーされてしまいます．
ですから，一本のプログラムとはいい難いわけです．

田口：変数類が全てクリヤーされてしまうのですから，後で自分で入れ直さなければならないのでしょうか．

鈴木：クリアされる変数を保存する手段として全部ファイルにしてしまうことができます．

15　アクセスとは何か

田口：ここまでに，アクセスといわれる言葉が聞かれますが具体的に説明して下さい．

鈴木：アクセスというのは，目的とするデータを読むとか書くとかする動作です．これは例で示した方がよくわかります．
たとえば，メモリがあります，メモリにアドレスが割り振られていて，それを目

印としてメモリを読むという事がアクセスしてということです．

アクセスするために，アドレスというものを出してそれによって，1つのメモリが選択されて，書くという動作を総称してアクセスといいます．

大野：アクセスにぴったりする使い言葉はないかな．

飯田：普通呼び出しといっているが．

鈴木：ディスクに対するアクセスとか，メモリに対するアクセスともいいますね．今の場合はディスクだからディスクのアクセスですね．ディスク中のデータはファイル単位ですから，ファイルをアクセスするという具合にもよく使います．

そこで，ディスク中のファイルをアクセスする為にどういう手段がとられるかという話になるわけです．

アクセスするというのは1つの目的とする，対象物が当然あるわけです．ですからディスク中に沢山ファイルが入っていれば，そのうちの1個が対象となります．対象を具体的に読み書きする為に，つまりアクセスする為にどういう手段がとられるかというのが1章，7章に書いてあるのです．

16　MOUNTとREMOVE

田口：プログラムやデータを打ち終わって，ディスクにSAVEしようと思い，はじめて，ディスクに電源が入っていなかったと気づいた場合やDisk　BASICの状態になっていなかったときの処置はどのようにすれば良いのでしょう．

大野：そういった場合には，カセットに一度プログラムをCSAVEを行なって退避してから，再びロードしてディスクにSAVEを行なう以外に手段はありませんね．

ディスクを使い慣れていない人におこりがちなミスです．

田口：mount　をするとき，ドライブ指定をしないで，mountをするとします．ドライブ指定がないので，両方ドライブがmountされようとしますが，なぜドライブ2の方からmountしようとするのか，ドライブ2にディスケットが入っていないと，どうなるのでしょうか．

飯田：どうしてドライブ2からマウントするか解かりません．ディスケットがドライブにマウントされているかどうかは，光学的に検出できるはずなのですが，今のところその検出は調べていません．したがってドライブにディスケットが入っていない状態でマウントしようとすると，たとえば，ドライブ2にディスケットが入っていないにもかかわらず，mount（ドライブ指定なし）を実行したりすると，いつまでも，ドライブ2のLEDが点燈したままになって，PC本体に制御が戻ってきません．あまり，勧められませんが，この状態で強引にディスケットを入れれば，マウントが終了します．

田口：　　　　　　の場合も同じ事が起きるのでしょうか．

飯田：remove　の場合は，mountされたディスケットに対して，　remove　が行なわれるので，心配はありません．

ただし，両ドライブを　mount　して，remove　をする時に，何かの理由でドライブ2からディスケットを抜いてしまったのでは，ドライブ2にディスケットが入っているとPC側が判断してしまうので，ドライブ2にディスケットが入るのを待ちつづける状態になります．

先ほど説明した mount の状態と同じよ

うな結果になってしまうのです．

このような初歩的なミスを行なわないように，マニュアルをよく読んでディスクの正しい使い方をしっかり覚えるのが賢明ですね．

田口： remove について，もう１つ聞きたいのですが，ディスケットにライトプロテクノッチをつけた場合では REMOVE できないのはどうしてでしょう．

大野：前にも説明があったと思いますが，REMOVE というのは，ディスケットに書き込みや SAVE を行なったのあとに，ディスケットの管理情報をディスケットにもどす命令なのです．

ライトプロテクトノッチをつけると書き込みができなくなりますから，この情報も書けなくなり REMOVE できないことになります．しかし，この管理情報は，ファイルを作ったり，削除したりしなければ変わることがないので REMOVE することができなくても全くさしつかえはないのです．

このようにライトプロテクトされているディスケットは，remove をせずにドライブから抜き取ってもかまわないのですが，次のディスケットを入れても MOUNT はできないことになります．前のディスケットを remove していないので Disk already mounted のエラーになるからです．ここで注意しなければならないのは，マウントされた状態になっているからといって，さしかえたディスケットに対して remove をしてはならないということです．このことについては７章でも触れていますが，さし替えてから remove を行なうと，前のディスケットの管理情報がさし替えたディスケットに書き込まれてしまいそのディスケットのほんとうの管理情報が破壊されてしまいます．

17　フォーマットとバックアップ

田口：次にディスケットに行なうものですが，ディスケットにフォーマットをかけるといいますが，かならず，フォーマッティングはいるものなのでしょうか．

飯田：また，トラックとセクタの話になってしまうのですが，トラックの区別はヘッドを移動することで可能です．問題はセクタはどうやって区別するのかになります．

これには，2 つの区別の仕方があります．ハードセクタとソフトセクタの 2 つに分けられるのです．

ハードセクタと呼ばれるのは，そのセクタがあるところに，インデックスホールと呼ばれる穴をそれぞれ開けておくのです．

たとえば，16個のセクタがあったら16個を中心に穴を開けておくのです．

その穴を光学的に検出したときにそのセクタの始まりだと判断できるのです．それに対してソフトセクタというのは，インデックスホールというのは，１つだけです．

そこから第１のセクタが始まると考えてください．

それから後のセクタの区切として特殊なマークをつけておくのです．

ここがセクタの始まりだよと印を付けておくのです．

それは，他の重要なデータと混ざらないように，特殊なビットパターンを入れてあるのが普通です．

現在のところ，ミニフロッピーは規格の統一がうまくなされていませんので，市販のディスケットをそのまま使用できるとはかぎりませんので，PC のドライブに

合うように，前に述べた，特殊なビットパターンを書き込む必要があります．
それがまず最初の第１段階のフォーマットになります．

大野：セクタの始まりの部分には，トラック番号とセクタ番号とが書き込まれます．このことによって，何らかのノイズでヘッドが誤まって移動してしまった場合でもそのエラーを検出することができるのです．
ようするに，「ぼくは，何トラックの何セクタです．」とディスケットに自己主張させるのです．

飯田：PC のドライブで使用可能にするには，さらに FAT や ID フィルードやディレクトリーを初期化する必要があります．
これがフォーマットのだいたいの内容です．

田口：ディスクのバックアップはフォーマットの次に大事なことだと思うのですが．

飯田：バックアップというのは，ディスケット中のデータのコピーのことで，ディスケットの良い所から悪い所までの全てをコピーしてしまうということです．ディスケット中のファイルが壊れることがたまにありますので，ファイルのコピーをしておけばその被害を最小限に食い止めることができるでしょう．

6　ディスクのハードウェア

フロッピーディスクは，1973年 IBM3740 データエントリシステムにおいて，紙テープやパンチカードに代わる新しい入力媒体として登場しました．フロッピーディスクは，従来の紙製媒体にない特徴を持っています．データ容量あたりの体積と重量が小さく，反復使用すると結局安くつくのです．これらは，カセットテープにもあてはまることですが，大きな違いは，ランダムアクセス可能であることです．これによって必要は情報がすぐに取り出せるので大幅に処理時間が短縮されました．

このように，入力媒体として登場したフロッピーディスクですが，その小形軽量の割に大容量，ランダムアクセス可能という特徴から，しだいにマイクロコンピュータの補助記憶として注目されてきました．そんな中で，1976年シュ一ガート社が，ミニフロッピーディスクを発表したのです．ミニフロッピーディスクは，前の標準フロッピーディスクより，ずっと小さいので取扱いがさらに手軽になりました．大きさが小さくなったので，当然容量も小さくなります．標準フロッピーディスクの容量は約400k，ミニフロッピーディスクの容量は約110kです．フロッピーディスクのもう一つの進歩は，倍密度化です．従来の単密度に比べて，倍密度による記録方法では，容量が2倍になります．したがって，標準フロッピーディスクで約800k，ミニフロッピーディスクで約220kの容量になるわけです．レコードに表と裏があるように，フロッピーディスクの裏表を使うことも可能です．このようにすると，さらに2倍の容量になるわけです．こうして，今日では両面倍密度が主流になりつつあります．記録方式，磁性面の改良によって今後さらに容量が増すことも考えられ，コストも，磁気バブル，カートリッジテープよりも，ずっと安いので，増々普及するものと考えられます．

図　標準ディスケットとミニディスケット

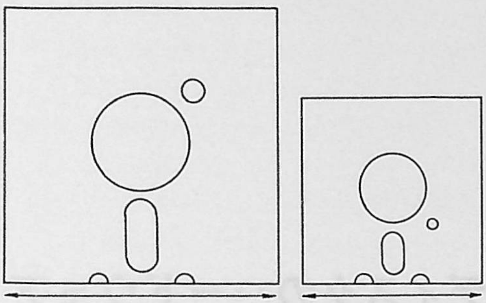


表　フロッピーディスクの種類

大きさ	標準（8.00×8.00インチ）
	ミニ（5.24×5.24インチ）
記録密度	単密度
	倍密度
セクタ分割	ソフトセクタ
	ハードセクタ
面利用	片面
	両面

6．1　フロッピーディスクの構造

フロッピーディスクは，フロッピーディスク本体とジャケットから構成されています．フロッピーディスク本体は，ドーナツ型のベースに磁性体を塗ったもので，この表面に情報が記録されています．中央の穴は，スピンドル孔と呼ばれ，ドーナツ盤レコードと同じように，ディスク本体を回転させるための軸が中に入ります．

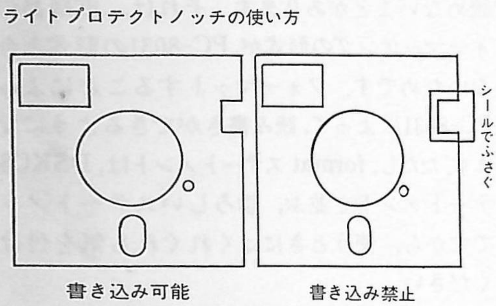
この穴によってディスクの中心，さらにトラックの位置を決めているわけですから，古くなって穴の縁がよれよれになると，エラーが発生しやすくなります．ディスケットが正しく入っていないのに強引にふたを閉めようとしても，この穴を傷めるので注意してください．ディスケット本体はその他に，小さな穴が，大きな穴の少し外側にあいています．この穴はインデックスホールと呼ばれ，円周方向の位置（後述するセクタのこと）を決定するのに使います．

ジャケットは本体より少し大きめのプラスチック製の入れ物で，同じように2つの穴があいています．その他に磁性体面にヘッドを接触するために，ヘッドアクセス孔と呼ばれる細長い穴があいています．このようにすることによって，ディスク本体の露出を必要最小限にしているわけです.ジャケットの内側には，不織布がはってあって，ジャケットと磁性面が直接接触するのを防いでいます．この不織布によって外部からの汚染による損傷や静電気によるデータロスからフロッピーディスク本体を保護しています. 外からはゴミが入りにくいかわりに，一度，中にゴミが入るとなかなか外には出てきません．こうなったら,いくら後でていねいに使っても，ドライブにかけるたびにキズを付けて，最悪の場合はヘッドを破壊してしまいます．ジャケットには，一部分切りこみがありますが，これはライトプロテクトノッチと呼ばれるもので，後述するフロッピーディスクドライブの，ライトプロテクトスイッチを動作します.この働きは，カセットテープなどとは逆で，何も付いていない状態が書き込み可能，ここをシールでふさぐと書き込み禁止となります．ジャケットには，2つのラベルが張ってあります．この2つのラベルの使い方は,PC-8031のユーザーズマニュアルに詳しく載っているのでそちらを見てください．

データが記録されるのは，ラベルの反対側の面です．この面はとくに大切にしてください．ドライブから出しているときはかならず付属の保護封筒に入れておいてください．もちろん，ディスク使用中は保護封筒にゴミなどが入らないように注意してください.PC-8031のフロッピーディスケット「取り扱い上の注意事項」は必ず読んでください．こういうと，かなり扱いにくいようですが，普通の使い方をしていれば十

分信頼性があります．

図　ライトプロテクスクノッチの使い方



6．2　トラックとセクタ

フロッピーディスクの磁性面は，半径方向に35本に分割されていて，これをトラックと呼びます．さらにトラックは，16個のセクタに分けられています．したがって，1枚のディスク上には，16×35＝560個のセクタが存在することになります．したがって，ディスク上のセクタの場所は，トラック番号と，そのトラックの何番目のセクタかを表わす．セクタ番号によって示されます．この2つの番号は，ちょうどディスク上の番地のようなもので，アクセスが効率良く行なえるようになっています．あるいは，ディスクを16×35の配列と考えてもよいでしょう．

1個のセクタには，256バイトの情報が入りますから，1枚のディスク上には，256×560＝14336バイトすなわち約143Kバイトの情報が入ります．

図　トラックとセクタ

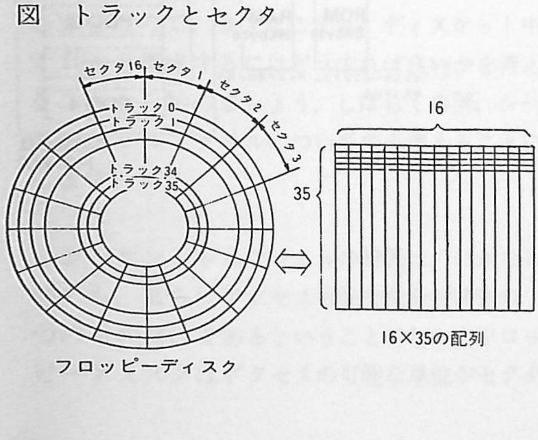
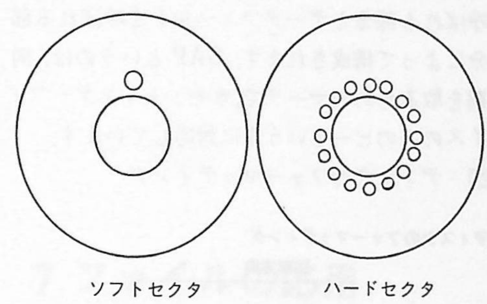


図　ソフトセクタ



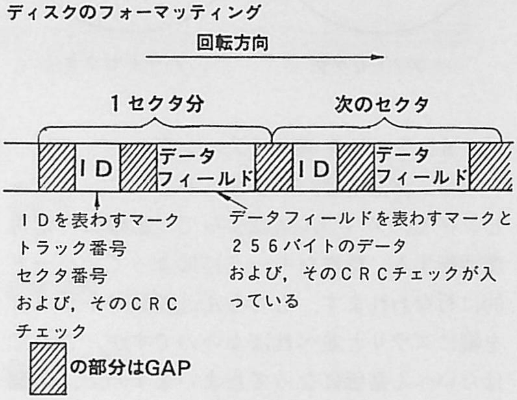
6．3　フォーマッティング

トラック指定は，後に述べるように，ステッピングモータ（パルスによって，正確に一定角度回転する，特殊なモータ）によって，ハード的に行なわれます．もちろん，35個のトラックを縦にズラリと並べればよいのですが，これではたいへん高価になってしまいますので，1個のヘッドをセッセと移動して使うわけです．それに対して，セクタの指定は，ディスクが回転しているので，そう簡単ではありません．トラックをセクタに分割する方法としては，ソフトセクタ方式とハードセクタ方式の2種類があります．ソフトセクタ方式というのは，セクタの切れ目に特殊なビットパターンを書き込んでおいて，実際にそのパターンをヘッドで読むことによって，セクタの位置を検出する方式です．これに対して，ハードセクタ方式というのは，ディスク上にセクタホールと呼ばれる小穴を必要なだけあけておいて，この小穴により，セクタの位置を光学的に検出する方式です．しかし，特殊な用途にしか用いられていません．

PC-8031では，ソフトセクタ方式を用いています．前に述べたように，ソフトセクタ方式では，特殊なビットパターンを，あらかじめディスク上に書き込んでおかなければなりません．この作業のことを，フォーマッテングと呼びます．

フォーマッテングは，インデックスホールを基準に行なわれます．　フォーマッティングによって，それぞれのセクタには，IDフィールドと呼ばれる部分とデータフィールドと呼ばれる部分によって構成されます．GAPというのは，同期を取るためのマークで，カセットインターフェイスのあのピーという音に対応しています．

図　ディスクのフォーマッティング



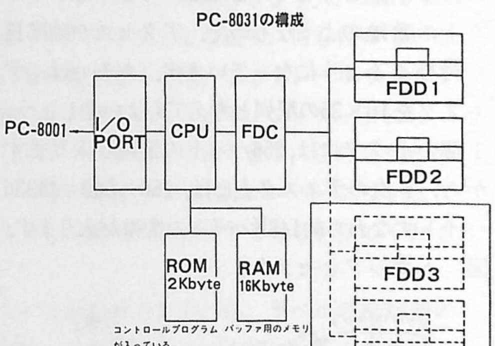
IDフィールドは，IDフィールドを表わすマーク，トラック番号，セクタ番号，及びそれらのCRCチェックが書かれています．CRCチェックとはサムチェックと同類のもので，そのデータに誤りがないか調べるための余分なデータです．このIDフィールドを読むことによって，今，ヘッドが，ディスク上のどの部分を読んでいるかを，知ることができるのです．データフィールドには，データフィールドを表わすマーク，256バイトのデータ本体，及びそれらのCRCチェックつまり，次に来るデータフィールドは読まなければならないのか，それとも無視するのか判断できるわけです．

ユーザーがデータを書き込むときは，このデータフィールドのみ書き変えて，IDフィールドは書き変えません．

以上のように，GAPと2種類のフィールドを書き込む手続きが，マニュアルに書いてあるフォーマッテングのSTEP 1に相当するもので，formatステートメントによって行います．市販されているディスケットは，そのままではPC-8031で読めないことがあります．それは，市販品のフォーマッテングの形式が，PC-8031の形式と合わないためです．フォーマットすることによってPC-8031によって，読み書きができるようになります．ただし，formatステートメントは，DSKO$ステートメントと並ぶ，恐ろしいステートメントですから，使うときにはくれぐれも気を付けてください．

しかし，このままではDisk BASICの通常のコマンド，ステートメントは使用することができません．それは，Disk BASICが，ファイルを管理するための情報，すなわち，FAT，ディレクトリ，IDの初期化を行なっていないからです．

これらも含めて，すべて自動的に行なってくれるのが，付属の，formatユーティリティプログラムです．



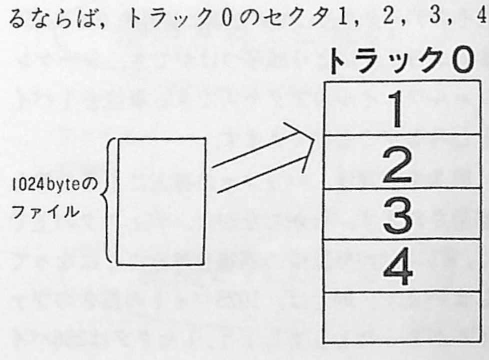
7 ファイルの応用

Disk BASIC で，新しく登場したコマンドやステートメントについては4章までで説明しました．本章では，ファイルの編成とその管理を理解することにより，各コマンドやステートメントが具体的にどのような働きをするかを考えてみることにしましょう．

7．1 ファイル構造の実現

6章でフロッピーディスク装置について説明しましたので，まずここでは，ディスケット中で file を構成するにはどうすれば良いかを考えることから始めましょう．しばらくの間，シーケンシャルファイルについてのみ考えることにします．

シーケンシャルファイルの特長は，いかなる時でも，直ちにアクセス可能な成分が特定の1つのものだけであるということでした．フロッピーディスクはアクセスの可能な単位がセクタでしたから，1つの成分が1セクタであり，いくつかのセクタがディスク上に列状にならんでいるとみなせば，テープと同様ディスクがシーケンシャルファイルの構造を持っていると，考えられることになります．列の順番づけの1番簡単な方法としては，トラックの番号の小さい順，同一トラック内ではセクタの番号の小さい順に，ファイルの領域を取ることで達成されます．たとえば，今ブランクディスケットがあって，長さが1024バイトのファイルを作ろうとするならば，トラック0のセクタ1，2，3，4

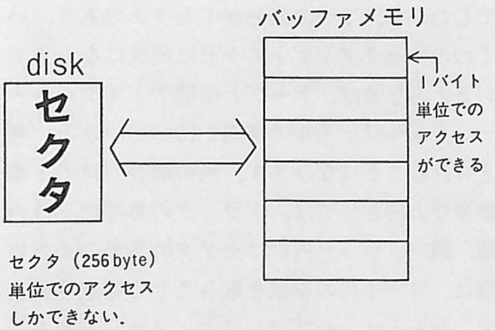


をこの順番で割り当てればよいのです．

さて，この構造のままでは，２つの問題があります．

1．アクセスできる成分が256バイト単位である．
2．ファイルの長さは256の整数倍でなければならない．

１の問題は，フロッピーディスクが１セクタ単位でのアクセスしかできず，その構成要素である１バイト，１バイトには直接アクセスできないということが原因です．したがって，１セクタ分の内容をメモリに移して，そのメモリ領域内でアクセスを行なうことによって，１バイトごとのアクセスが実現されます．このためのメモリ領域をバッファと呼び，BASICと，ディスクとのデータのやりとりは全てバッファを介して行なわれます．



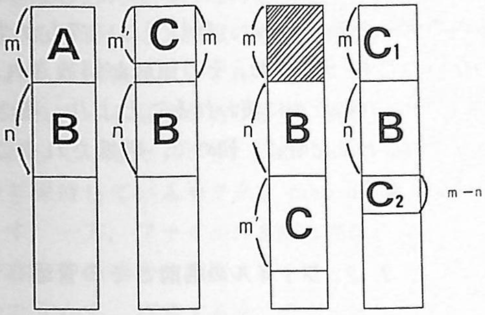
このようにして，セクタ内のデータに対してもそのデータの入っている順に番号を０から255までふることにより順序づけができ，シーケンシャルファイルのアクセスできる単位を１バイトとみなすことができます．

第２の問題は，バッファの導入により自然と回避されます．しかしながら，ディスクの上では，常に256の整数倍の領域を取ることになってしまいます．例えば，1025バイトの長さのファイルがあったとしましょう．１セクタは256バイトですから，

1025÷256＝４　余り　１

つまり，４セクタでは，１バイト分の容量が足りないので，５セクタが必要になるわけです．５セクタ目の256バイトのうちの最初の１バイトは，こうして使われますが，残りの255バイトは，実際には使ってなくても，使ったことになってしまいます．このことは，大変重要な事実です．つまり，ディスクの容量が何10Kバイトもあると言っても，実際にはその全てを有効に使えるとは限らないのです．明らかに，容量を最も有効に使うのは，ファイルの長さが，256の整数倍の時です．しかしながら，実際には，PCではディスケットの記憶領域をさらにクラスタという単位で管理しているために，２Kバイト＝2048バイトの整数倍でないと完全に有効には使えません．このことについては，後でもう一度述べます．

さて，話を進めましょう．先ほどのファイルの領域のとり方は，ファイルが論理的に連続したものであるから，領域も物理的に連続した領域（つまり互いに隣接したセクタ）に取ろうというものでした．このままの方法では，複数のファイルがある時には，領域の使用効率の点で問題が生ずることがあります．例えば，図７-３ａのように大きさがそれぞれ，m，nの２つのファイルA，Bがあったとしましょう．ファイルAが不要になって，大きさがm′のファイルCを代わりにしまうとすると，m′≦mの場合には図７-３ｂのようにして，Cをしまうことができますが，m′>mの場合には，ファイルCに対して，Aのあった場所を割り当てることはできません．そのためにどうしても，図７-３Cのように割り当てざるをえません．つまり，不要になったAの領域を使えないのです．



7-3a 7-3b 7-3c 7-3c'

この問題は，領域を図７－３C′のように２つに分けてとれば解決できます．つまり，物理的には，連続でない領域を，論理的には連続しているとみなすことができるような工夫が必要となります．そこで，ファイルを，セクタの連結で表現する方法を考えて，シーケンシャルなアクセスは，この連結の情報を参照して，次の成分（セクタ）の位置をみつけて行なうことを考えてみましょう．図７－３ａでＡ，Ｂ，Ｃの各ファイルの大きさが，セクタ数にしてそれぞれ４，５，６であったとすると，各ファイルの領域の連結の情報は次のようになるでしょう．

A	B
0－1	0－5
0－2	0－6
0－3	0－7
0－4	0－8
	0－9

トラックの番号　セクタの番号

図７－３C′では不要になったＡの領域がＣによって使われるためにＣの情報は次のようになります．

C
0－1
0－2
0－3
0－4
0－10
0－11

このように，連結の情報を導入することにより，物理的には連続してないトラック，セクタの情報であっても，論理的には１つの連続したファイル中の情報であるとみなすことができるようになるのです．

7．2　ファイルの管理

7．1節では，ファイルの実現法を考えましたが，その際に付随して必要となる問題が２つあります．

1．ディスケットの記憶領域の使用管理をどのようにして行なうか？
2．各ファイルの識別を行なうにはどのようにすればよいか？

１の問題はデイスケットのどこをすでに使用していて，どこをまだ使用していないかを知るために是非とも必要な事柄です．２は，ファイルの名前による識別をデイスケット上でどのように実現するかという問題です．ここではこの２つの問題について考えてみましょう．

7．2．1　デイスケットの記憶領域の管理

7．1節で考えたように，ファイルの生成に伴なう領域の割当てと，消去に伴なう使用していた領域の解放を実際に行なうにはどのようにすれば良いでしょうか．各セクタの状態には使用されている，使用されていないの２つがあることはすぐにわかります．ファイルの生成と，消

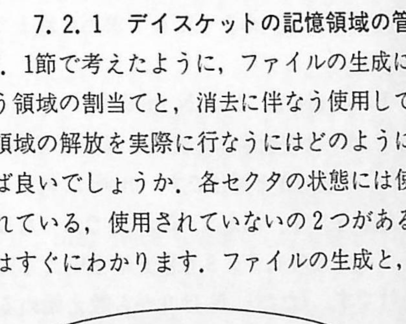


図７・１－１

去を行なう場合にこの状態の遷移が起こることは明らかです．このように，各セクタの状態が2つであるので，この情報は1ビットで表現することができます．使用されていない状態を0で表わし，使用されている状態を1であらわすとすると，ファイルを作るときに0から1に，消去する時に1から0に変えれば良いことになります．1ビットで1セクタの使用状態を表わすことにすると，1枚のディスクには35個のトラックがあり，各トラックには16のセクタがありますから，16×35＝560 bit＝70byte の情報が必要となります．このようにして，領域の使用状況を表現する方法を bit-map 型式と呼びます．この方法によれば，ファイルを生成する際にどのトラック，セクタを割り当てれば良いかも容易に求めることができるようになります．すなわち，bit-map を最初から調べていき，最初に出会った0であったビットを1に変えて，その bit が何番目の bit であるか数え，トラック，セクタの番号を求めれば良いわけです．ビット番号をN，トラック，セクタの番号をT，Sとすれば，T, S は次のようにして求められるでしょう．

$$T=N \text{ div } 16 \quad S=(N \text{ mod } 16)+1$$

ここで，div は整数除算を表わす演算子で PC の BASIC で使う￥と同じです．mod は剰余を求める演算子です．例えば，N が19であったならばトラック1のセクタ3を使えばよいことになるわけです．(ただし N は0から数え始める)

逆にファイルを消去する際には，連結の情報から，そのファイルの使用している T と S を求め，　16＊T＋S　の式で求めた N 番目のビットを1から0に変えてやることにより，ファイルの消去が実現されます．ファイルの消去は，その使用していた領域を解放するだけで，そのファイルの保持していたデータ自身は実際には残っていることに注意して下さい．7．2．2節で述べるように，ファイルの名前を消去することにより，そのファイルのアクセスが不可能になるのでデータそのものは，残しておいてもかまわないのです．データは，その領域が解放され，他のファイルのために使われることより，書き込みが行なわれた時に，初めて，破壊されることになります．

7．2．2　ファイルの名前とその管理

7．1節でファイル構造の実現方法，7．2．1節でその使用領域の管理方法を考えてみました．ここでは，BASIC が行なうようにファイルを名前で参照するにはどのようにすれば良いかを考え，1つの方法を示しましょう．(ここで考える方法は，1つの実現方法であって，PC の DISK BASIC で使われている方法ではありません．PC で行なっている方法については7．3節で述べます．)

7．1節で考えたように，ファイルはその構成要素であるセクタの連結で表現されます．したがって，ファイルを名前で参照できるようにするには，名前と連結の情報とを結びつけてやれば良いことになります．ABC という名前のファイルで長さが n セクタを必要とするものがあったとすると，図7．2-1のように表わすことができるでしょう．ファイルの名前によって，その連結の情報を参照し，さらにその情報によって，ファイルのデータがどのトラック，セクタにある

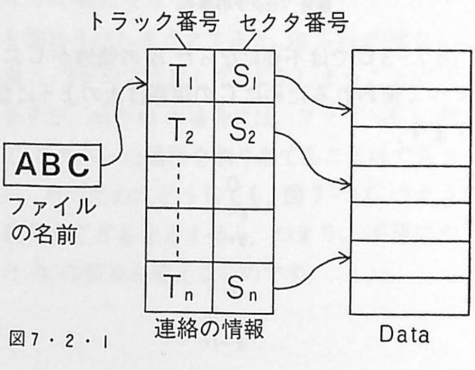
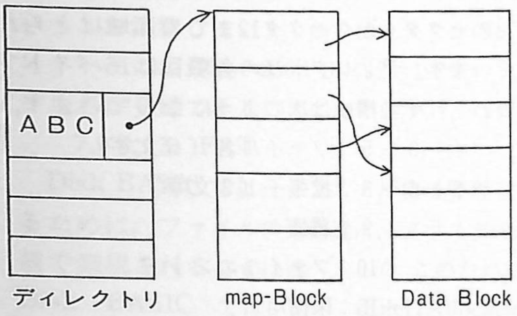


図7・2・1

かを知るわけです．このように，ファイルを構成する情報には３種類のものがあるわけですがこれらの情報は，いずれもディスケット上に記憶されている必要があります．実際にデータを保持しているセクタを，Data block．連結の情報を保持しているセクタを map block と呼びます．一方，ファイルの名前は複数のファイルが存在する場合に識別する手段として用いるものですから，存在する全てのファイルの名前を表にしておくのが便利です．このための領域をディレクトリ（directory）と呼びます．ディレクトリには，ファイルの名前とともに，そのファイルの使用している領域を知るために，map block のあるトラックとセクタの番号も入れておきます．すると，ファイルの参照は次のようにして達成されます．

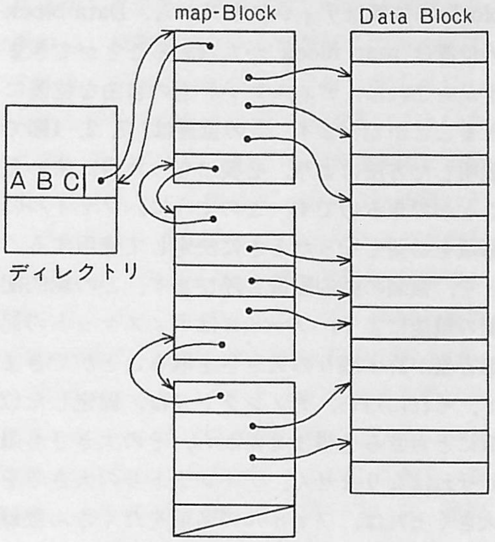


1．　ディレクトリの中から，参照しようとするファイルの名前を探す．
2．　ファイルの名前が見つかったら，名前といっしょに記録されている，map block の所在地（トラックとセクタの番号）をみて，そのセクタの情報（連結の情報）を読み込む
3．　読み込んだ連結の情報を参照して実際のデータのある場所（Data block の位置）を調べデータに対してアクセスする．

このように，ディレクトリは，台帳という意味が示すように，ファイルの名前とその所存地に関する情報を保持しているのですから，ディレクトリ自身のディスケット上での位置は固定していなければなりません．それに対し，map block の位置はディレクトリから，Data block の位置は map block から求めることができますからこれは，ディスケット上の自由な位置にとることができます．この位置は，7．2．1節で説明した方法により，必要になった時に決めることができるのです．このように，ファイルの領域を必要となったときに決定して使用することを，領域の動的配置と呼びます．この動的配置の機能により，ファイルはディスケットの記憶容量が許す限りの大きさを取ることができます．それに対し，ディレクトリは，固定した位置にとらざるを得ませんから，その大きさも限定せねばなりません．ディレクトリの大きさを大きくとれば，ファイルの名前をたくさん登録することができますが，Data の領域は，それだけ小さくなってしまいます．したがって，ディレクトリの大きさは，１枚のデイスケットにいくつのファイルを作れるようにするかによって決められます．

map block の大きさは，ファイルの長さによって決まるという性質を持っていますから，長さが可変であるような構造を持たせねばなりません．そのために，ファイルを連結で表現したように，map block も必要ならば連結を行なわねばなりません．すなわち，１つのファイルで使用する map block が１セクタ分だけであると，トラックとセクタの番号を覚えるのに１バイトずつ消費するために，256÷２＝128の長さの連結の情報しか表現できません．つまり，128セクタの長さまでのファイルまでしか表現できないことになります．ディスケットには16×35＝560セクタがありますから，これだけでは不十分です．この解決策として，１セクタの map block ではたりない場合には，map block の連結を行なう情報を持たせることになるわけです．通常

この連結は，双方向性のリストにより実現されます．



このような構造を持たせることにより，1セクタのマップで足りなくなった時には，次のマップを参照することによってファイルへのアクセスが可能となるわけです．

7．3　Disk　BASICでのファイル管理

7．2節までは，ファイルの管理方法として1つのモデルを考えてみましたが，ここではPCのDisk　BASICが実際に採用している方法について説明します．

7．3．1　クラスタ

いままでは，ファイルの管理の単位をセクタ単位で考えてました．そのために，ファイルの長さが長くなると連結の情報の量もずいぶんと多くなってしまいます．Disk　BASICでは，8セクタを1つの単位として管理を行なっています．この単位をクラスタと呼びます．そのために，ファイル構造はクラスタの連結として表現されるようになります．また，ファイルの領域もクラスタ単位でとられます．1セクタは256バイトでしたから，1クラスタは2Kバイトになり，0トラックの1セクタから8セクタまでを0クラスタ，34トラックの9セクタから16セクタを69クラスタとするクラスタ番号をつけて管理を行なっています．

この方法によれば，ファイルの列構造を表現する連結の情報が少なくて済みます．しかし，領域の管理の面からみると，最少の管理単位が2Kバイトであるので，どんなに短いファイル（たとえ1バイトであっても）でもデイスケット上では2Kバイトを消費することになってしまいます，物理的には2Kバイトの整数倍の領域しかとれないことになります．

7．3．2　デイレクトリ

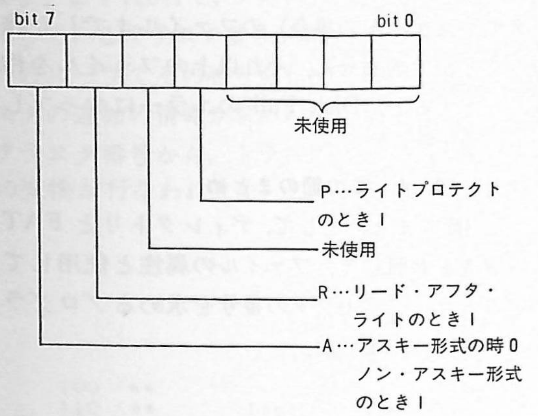
Disk BASICではディレクトリは，トラック18のセクタ1からセクタ12までの領域にとられています．ディレクトリの各項目は16バイトで表わされその構成は次のようになっています．

バイト0～5：ファイル名（6文字）
バイト6～8：拡張子（3文字）
9：属性
10：ファイルエントリ
11～15：未使用

ファイル名と拡張子はファイルの名前の識別に使われるものです．属性は，そのファイルの持っている属性を保持しています．属性にはアスキー型式，リード・アフタ・ライト，ライトプロテクトの3種類があり，次のように表わされています．

3種の属性のうち，RとPは，SETによって付けられ，Aは，SAVEコマンドによってつけられます．

ファイルエントリは，そのファイルの使用している最初のクラスタ番号を持っています．7．2節までのモデルでの最初のmap blockの位置



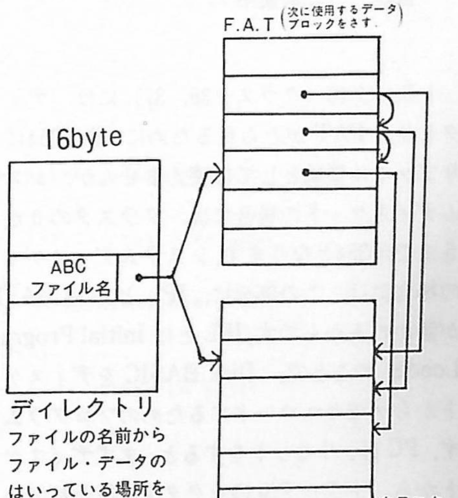
の情報に相当するものですが，Disk BASIC では，map block と領域管理のための bit-map を一緒にして，次に説明する FAT で表現しています．

このようにして，1つのファイルあたり，16バイトを消費していますので12セクタ分のディレクトリには最大　256÷16×12＝192個のファイル名が登録できることになります．

7．3．3　FAT

Disk BASIC では，クラスタ単位で管理をするために，ファイルの列構造は，クラスタの連続で表現されることは述べました．このために，Disk　BASIC　では　FAT (File　Allocation Table) という領域をとって，連結の情報を保持することにしています．FAT は，ディスケット上のトラック18のセクタ14からセクタ16までの領域に置かれていますが，この3セクタの内容は全て同じになっています．各セクタの256バイトのうちの70バイトで FAT は表現されているのですが，安全のために，同じものを3つ持っていて，3つが一致しないとそのディスケットは使えないようにしているのです．

FAT は70バイトの配列として表現されていますが，この70バイトという大きさは，デイスケット上に 0 から69までの70のクラスタが存在するという事実と符号しています．ファイルの最初に使用しているクラスタの番号はディレクトリ中にあることはすでに述べましたが，この番号を N1とすると，配列 FAT の FAT (N1) なる要素には，その次に使用するクラスタの番号 N2が入っているのです．同様にして FAT(N2) には3番目に使用するクラスタの番号が入っているという具合にして，ディスケット上の全てのファイルの連結の情報が FAT の70バイトだけで表現されているのです．後続するクラスタがない場合には，そのクラスタ内の8セクタのうちの実際に使用されているセクタ数m（1≦m≦8）＋192なる数が入っています．この情報によって，最後のクラスタの何番目のセクタまで達すればファイルが終わりかがわかります．192足しているのはクラスタ番号と区別するためです．



このように，FAT はファイルの列構造の表現を行なっていますが，また同時に領域の管理も行なっています．すなわち，使用していないクラスタに対応する FAT の要素には，値として255を持たせ，使用しているクラスタと使用していないクラスタとを判別しているのです．FAT の要素がとる値を区分すると次の表のようになります．

要素の値	対応するクラスタの使用状態
0～69	対応する8セクタ全てを使用中．後続するクラスタを持ち，値はそのクラスタ番号を示す．
193～200	対応するクラスタはファイルを構成する最後のクラスタであり，値から192を引いた数が，実際に使用されているセクタ数である．
254	Disk BASIC が使用しているクラスタであり，ファイル領域としては使用できない．ディレクトリや FAT 自身の入っているクラスタがそうである．
255	未使用

トラック18（クラスタ36，37）には，ディレクトリと FAT がとられるために，値は254になりファイル領域としては使えませんが，システムディスケットの場合には，クラスタの0から5までが254となります．システムディスケットの場合には，この部分に，IPL と Disk BASIC が置かれるからです．IPL とは Initial Program Loader のことで， Disk BASIC をディスケットからメモリへロードするためのプログラムです．PC は，リセットをすると，まずディスケットから，トラック0のセクタ1に置かれている IPL を読み込み，その IPL によってトラック0のセクタ2からトラック2のセクタ16までにある Disk BASIC を読み込むのです．このように，ディスケット上には，BASICが使用するクラスタがあるので，ユーザがファイル領域として使えるのは，データ・ディスケット（Disk BASIC を含まないもの）で68クラスタ，システム・ディスケットでは62クラスタになります．ですから，ディレクトリには，192までのファイル名を登録できるようにはなっていますが，少しでも長さのあるファイルならば，68ケ（データディスケットの場合）のファイルまでしか作ることができません．それ以上のファイルを作ろうとすると，Disk Full のエラーになってしまいます．

7．3．4　この節のまとめ

7．3節のまとめとして，ディレクトリと FAT の構造を参照して，ファイルの属性と使用しているトラック，セクタの番号を求めるプログラムを作ってみました．

このプログラムについて一応の説明をしておきましょう．

240行でファイルの名前を入力します．名前の入力型式はロードやセーブをする時と同じでドライブの指定もつけることができます（〝でくくる必要はない）．入力した名前から，260行でドライブの指定の部分を取り出し，280行から310行までのループでファイル名と拡張子を取り出して，ディレクトリとの比較ができる形にします．

サブルーチン SEARCH は，指定されたファイルがはたしてディレクトリ中に登録されているかを調べます．ディレクトリは，DSKI$ により1セクタずつ読みとられ，文字変数 DIR$ の中に一項目ずつ入れられます（450行）．DIR$ の要素は，ファイル名を持つ N$ と比較され，名前のサーチが行なわれます．見つからなかった時には，セクタの指定 S を1つ増やし，次のセクタ内を調べます．

サブルーチン PRINT　OUTは，指定されたファイルの属性と，使用している領域を求め印字します．ディレクトリ中にあった情報（N$）の10番目の文字は属性でしたから，580行で K に属性が得られます．

660行から690行では，配列 FAT%に DSKI$ を使って FAT の内容を読み込んでいます．

710行から810行は FAT の情報をもとに使用しているトラックとセクタの番号を求め印字し

ます．まず710行で，ディレクトリから最初に使用しているクラスタ番号が求められ，740行～810行のループでその番号をもとに FAT 中のクラスタの連結の情報が求められていくと同時に，クラスタ番号から，トラック，セクタの番号への変換が行なわれます．このループ内では変数 GT が使用したセクタ数をカウントするのに用いられていますので，最後に820行でプリントを行ないます．関数 FNA は変数の値の書式を整えるためのもので PRINT　USING の代わりに使っています．

プログラム

```
100 '**
110 '**         Demo Program for Section 7
120 '**
130 '**  This program print file attribue & used Track/Sector number
140 '
150 '*
160 '*     MAIN ROUTINE
170 '*
180 '
190     CLEAR 1000                                      'Reserve string space
200     DIM FAT%(70)
210     DEF FNA$(X)=RIGHT$("     "+STR$(X),5)
220     FIELD #0,255 AS A$
230 '
240     INPUT "File name:";NA$:PRINT                    'Input file name
250     PRINT "File informations for ";NA$
260     IF MID$(NA$,2,1)=":"
          THEN D=VAL(LEFT$(NA$,1)):NA$=MID$(NA$,3,LEN(NA$)-2)
          ELSE D=1
270     N$=""
280     FOR I= 1 TO 9                                   'Get file name
290       N1$=MID$(NA$,I,1)
300       IF N1$="." THEN N$=N$+SPACE$(7-I)
            ELSE IF N1$="" THEN N$=N$+" " ELSE N$=N$+N1$
310     NEXT I
320     N$=LEFT$(N$,9)
330 '
340     GOSUB 390                                       'Search file
350     IF FIND THEN GOSUB 550
             ELSE PRINT "File not found"
360     END
370 '
380 '*
390 '*    SUBROUTINE:  SEARCH
400 '*
410 '
420     S=0:FIND=0:LAST=0
430 '
440     S=S+1
450       D$=DSKI$(D,18,S)                              'Read directory (1 sector)
460       DIR$=A$                                       ' into DIR$
470       P=INSTR(DIR$,N$)                              'Search file name
480       IF P<>0 THEN N$=MID$(DIR$,P+9,2):FIND=-1
490       IF ASC(MID$(DIR$,249,1))=255 THEN LAST=-1
500     IF NOT FIND AND NOT LAST THEN GOTO 440
510 '
520 RETURN
530 '
```

```
540 '*
550 '*    SUBROUTINE:  PRINT OUT
560 '*
570 '
580     AT=ASC(LEFT$(N$,1))                              'Get file attribute
590     PRINT :PRINT "ATTRIBUTE:";
600     IF AT<128 THEN PRINT "A ";                       'Decode attribute
610     AT=AT MOD 128
620     IF AT>63 THEN PRINT "R ";:AT=AT-64
630     IF AT=0 THEN PRINT ELSE PRINT "P"
640     PRINT
650 '
660     D$=DSKI$(D,18,14)                                'Read F.A.T
670     FOR I=0 TO 70                                    ' into FAT%( )
680       FAT%(I)=ASC(MID$(A$,I+1,1))
690     NEXT I
700 '
710     C=ASC(MID$(N$,2,1))                              'Get file entry
720     PRINT "  ｸﾗｽﾀ ﾄﾗｯｸ  ｾｸﾀ"
730     CT=0                                             'Clear Sector count
740     C1=FAT%(C)
750       IF C1<70 THEN J=8 ELSE J=C1 MOD 16
760       T=C¥2:S=(C MOD 2)*8+1:CT=CT+J
770       FOR I=1 TO J
780         PRINT FNA$(C);FNA$(T);FNA$(S)
790         S=S+1
800       NEXT I
810     IF C1<70 THEN C=C1:GOTO 740
820     PRINT:PRINT CT"Sector Used"
830 '
840 RETURN
```

実行結果

```
run
File name:? demo

File informations for demo

ATTRIBUTE:

  ｸﾗｽﾀ ﾄﾗｯｸ  ｾｸﾀ
   38   19    1
   38   19    2
   38   19    3
   38   19    4
   38   19    5
   38   19    6
   38   19    7
   38   19    8
   40   20    1
   40   20    2
   40   20    3

 11 Sector Used
Ok
```

```
ile name:? fdump

File informations for fdump

ATTRIBUTE:A P

  ｸﾗｽﾀ ﾄﾗｯｸ  ｾｸﾀ
   39   19    9
   39   19   10

 2 Sector Used
Ok
run
File name:? prin25.asc

File informations for prin25.asc

ATTRIBUTE:A

  ｸﾗｽﾀ ﾄﾗｯｸ  ｾｸﾀ
   32   16    1

 1 Sector Used
Ok
```

7．4　ファイル構造とコマンド，ステートメントとの関係

7．3節までで，ファイルの構造とその管理方法がわかりましたので，ここでは，それらとコマンドやステートメントとの関係を考え，具体的にどのような働きをしているのか説明しましょう．

7．4．1　MOUNT，REMOVE

mount　はディスクの使用開始宣言でした．mountにより，BASICは指定されたドライブ中のディスケットのFATをメモリ中にロードします．FATはファイルの参照の度に必ず必要となる情報ですから，メモリ中にロードしておくのです．PCでは，ドライブは4台までつなげることができますので，FATをロードして使うメモリ領域も4台分準備されており，決められた場所にロードされます．

一方 remove は，メモリ中のFATをディスケットに書き込む仕事を行ないます．新しくファイルを作ったり，書き込みを行なった時には，ディスケットの使用状況を表わすFATの内容も変化しているはずですから，ディスケットをドライブから取り出す前には，removeにより，FATの内容の更新を必ず行なわなければならないのです．mount したディスケットに対して読み出ししか行なわなかった場合には，FATの内容は全く変化しませんから，removeをせずにディスケットをドライブから取り出したりリセットをかけたりしても大丈夫です．しかし，remove をしないでディスケットを差し換えて使うと，メモリ中にあるFAT　はもとのディスケットのFATですから，正しくファイルを扱うことができなくなってしまいます．

removeのし忘れによってどのような事故が発生するかを考えてみましょう．

1．mount したディスケットに対して読み出ししか行なわなかった状態で別のディスケットと入れ換えて remove した時．

2．mount したディスケットに対して書き込みも行なった状態で別のディスケットを入れかえて remove した場合．

1の場合には，もとのディスケットに対しては，FATの内容は変化していないのですから，何も問題はありません．一方，後で入れ換えたディスケットの方は，removeにより，もとのディスケットのFATの内容が書き込まれてしまい本来のFATの情報はこわされてしまうことになります．つまり，入れ換えた方は使いものにならなくなってしまいます．

2の場合には，1と同様の理由から入れ換えた方のディスケットはダメになります．一方もとのディスケットには，到命的な損害はありませんが，書き込みを行なった情報は，実際にはディスケット上に記録されているにもかかわらず．それを参照するために必要なFATの情報がないために書かれなかったと同じことになってしまいます．

7．4．2　NAME，KILL，SET

ファイルの名前が登録されているのは，ディレクトリでしたから，ファイル名のつけかえを行なうNAMEは，ディレクトの書きかえを行なうことで実現されるのは簡単にわかるでしょう．それに対し，KILLでは，ファイル名の削除と使用していた領域の解放を行なう必要がありますから，ディレクトリとFATの両方に働きかけることになります．まず，ディレクトリ中から，KILLするファイルを探し，その項目を0にします．この0はKILLしたファイルだという目印です．FATの方はディクトリからのクラスタ番号をたどっていき，対応する要素を255にすることで，領域の解放が達成されます．SETによるファイル属性の設定も，ディレクトリへの書き込みを行なうだけです．

7．4．3　OPEN, CLOSE

OPEN はファイルの使用を可能にするために次のような仕事を行ないます．まずディレクトリを調べ，ファイルが見つかった場合には，そのファイルに関する情報（ファイルエントリと属性）をメモリ中のファイル　ディスクリプタと呼ばれる場所にセットします．ファイルがない時には，モードに応じて，エラーを出すか，新しくファイルを作ります．また，バッファの割り当ても OPEN の重大な仕事の1つですが，PC の Disk　BASIC ではファイル番号の指定が，そのままバッファの割り当てに相当します．ファイルディスクリプタとは，ファイル操作のために必要な情報を保持しておくための領域でバッファと連続した領域にとられます．その内容の基本的なものは次の5種類です．

●File entry

これは，ディレクトリ中のものをコピーしたもので，そのファイルの使用する領域がどこのクラスタから始まるかを示すものです．7．3節のプログラムでも示したように，この情報があれば，ファイルの大きさも知ることができます．そこで，LOF 関数の値を求める場合に用いられます．

●Next R／W block

次に読み書きすべきトラック，セクタの番号を保持します．入力の際にバッファが空である場合または，出力の際にバッファが一杯になっている場合には，この情報の示すトラック，セクタに対して読み書きが行なわれるわけです．PC の場合には，クラスタ番号と，クラス内の位置（0～7の数）の2バイトで表現されます．ディスクに対する読み書きが行なわれる度にクラスタ内の位置が1つ増やされ，7まで進むと0に戻ってクラスタ番号が1つ増やされます．この2つの情報からトラックセクタ番号への変換のしかたは，7．3節のプログラムにあるとおりです．この情報は，LOC 関数の返す値に相当するものです．

LOC 関数の値＝クラスタ番号＊8

＋クラスタ内位置

OPEN 動作はこの情報の初期設定を行ないます．すなわち，APPEND のモードでは file entry をたどり，ファイルの終りを見つけて，それに続く値をセットし，その他の時には file entry の値そのものになります．

●Next　R／W　byte

次に読み書きをするバッファ内の位置を示します．バッファは1セクタ分，256バイトですからこの情報は1バイトで表わされます．OPEN 時には，0にセットされ，PRINT やINPUT があった場合には，この情報の示すバッファ上の位置に対して入出力を行ない，読み書きを行なったバイト数分だけ値を増やします．値がオーバーフローした時は，バッファが空（入力時）または一杯（出力時）になったということですから，バッファの内容を Next R／W　block の示すディスク上アドレスに読み書きします．

●属性

バッファの属性を保持します．OPEN 時にはディレクトリ中の属性をコピーし初期設定します．SET によってバッファに属性がつけられたり，変更された場合には，この情報が変化するわけです．

●モード

ファイルがどのモードでオープンされているかを示す情報で，INPUT，OUTPUT，APPEND，ランダムの5つの状態のいずれか1

つを示します．OPEN 時には，そのモードの指定に応じて後者4つのいずれかの状態にセットされ，クローズ時にクローズ状態にセットされます．入出力を行なう時は，このモードがクローズ状態でないことを確めてから行なわれ，クローズ状態の時には File not open のエラーがだされるわけです．

CLOSEは，ファイルの使用していた，バッファの後片づけを行なうものです．すなわちバッファ内を全てクリヤし，ファイルディスクリプタのモードをクローズ状態にセットします．ファイルが出力用にオープンされていた場合には，バッファをクリヤする前に重用な仕事があります．1つは，バッファ内の情報のはき出しです．出力時には，バッファが一杯になるまではディスクへの書き込みは行なわれませんから，CLOSEする時には，バッファにはディスクに書き込まねばならないデータが残っていることがあります．したがって CLOSE ではこれを書き込んでやらねばなりません．また同時にファイルの終わりの目印をつけてやる仕事も必要です．これは，入力時に EOF のチェックを行なうために是非とも必要です．結局 Disk BASIC では CLOSE 時に次のような処理を行なうことになります．

ファイルに書き込みをしていなければバッファをクリヤして，モードをクローズ状態にして処理を終る．

書き込みをしたのであれば，Next R／W byte の示す位置に EOF の目印として16進で1Aを書き込み，バッファの残りを全てクリヤする．次にこのバッファの内容を Next R／W block の示すディスク上アドレスに書き込む．最後にバッファをクリヤし，モードをクローズ状態にして処理を終る．

7．4．4　PRINT#，INPUT#

もうすでに述べたように PRINT# はバッファへの出力，INPUT#はバッファからの入力を行ないます．INPUT#でバッファの内容を全て読んだときは，Next R／W block の示すディスク上のアドレスからバッファへの読み込みで行ないます．Next R／W block の値の更新は，クラスタ内位置を1つ増やすことによって行なわれますが，7を越えた時には FAT を参照して次のクラスタ番号を求めます．

PRINT#の時には，バッファが一杯になった時点でディスクへの書き込みを行ない，クラスタ内位置が7を越えた時には FAT を調べて，未使用クラスタを探しだし，Next R／W block にセットする仕事を行なうことになります．

7．4．5　PUT#，GET#

PUT，GET はバッファの状態に関係なくディスクへの出力を行なうと考えれば，PRINT#，INPUT#と同じようなことだということがおわかりになるでしょう．レコード番号が指定された時には，file entry からレコードの数ぶんだけ FAT 内のファイルの連結の情報をたどっていけばよいわけです．

——APPENDIX A——

エラーチェック　リスト

ここではディスク関係のエラーの意味とその主な発生原因を挙げておきます．その他のエラーについては PC　#1を参照してください．

ERR コード：50　Field　overflow

意味：FIELD 文で256バイト以上の文字を割り当てた．

原因：1つの FIELD の大きさは256バイトであるのに対し，FIELD 文での割当ての合計がこれを越えている．

──→ フィールドの構成を変更して256バイト以内におさめて，または，複数の FIELD 文を用いて処理することを考える．

ERR コード：51　Internal　error

意味：BASIC 内部でのエラー

原因：通常このエラーは発生しない．

ERR コード：52　Bad　file　number

意味：ファイル番号がまちがっている．

原因：指定したファイル番号が0から BASIC の起動時に定めた上限までの間にない．

：シーケンシャルファイルでのファイル操作にファイル番号0を使おうとした．

ポイント：BASIC の起動時の上限に指定をせずにリターンするとファイル数は5までに制限される．0番は DSKI$，DSKO$ でのみ使うことができる．

ERR コード：53　File　not　found

意味：指定されたファイルが見つからない

原因：OPEN，LOAD，KILL 等で指定したファイルが存在しない．ファイル名は大文字，小文字も区別することに注意．ドライブの指定を忘れることも多い．

ERR コード：54　Bad　file　mode

意味：ファイルのモードがまちがっている．

原因：非アスキー形式のファイルを MERGE しようとした．

──→ アスキー形式で SAVEしなおす．

(例) SAVE "ファイル名", a

ERR コード：55　File　already　open

意味：指定されたファイルはすでにオープンされている.

原因：同一のファイルを２回オープンしようとした.

：すでに使用中のファイル番号を指定してファイルをオープンしようとした.

：オープンしているファイルを KILL しようとした.→CLOSE する.

ERR コード：56　Disk　not　mounted

意味：指定されたディスクは MOUNT されていない.

原因：MOUNT のし忘れ.

：ファイル名中でディスク指定を忘れている.

ERR コード：57　Disk　I/O　error

意味：ディスクへの入出力中にエラーが生じた.

原因：ディスクへの入出力の際には，エラーが生じても，BASIC は10回までの再試行を行なってくれます．それでもエラーが回復不可能な場合にエラーとなります．このエラーが出るようになったディスケットはフォーマッティングをし直すか，使用時間が多ければ廃棄処分にした方が良いでしょう.

ERR コード：58　File　already　exists

意味：指定されたファイル名はすでに存在している.

原因：NAME でファイル名を変更するさいに指定された新ファイル名と同じ名前を持つファイルがすでに存在している.

───→ ファイル名を変更する.

ERR コード：59　Disk　already　mounted

意味：指定されたディスクはすでに MOUNT されている.

原因：１つのディスケットを複数回マウントしようとした.

：ディスケットを入れ替えて mount しようとした場合は，前に使用していたディスケットを remove し忘れているということである.

ポイント：remove のし忘れの場合もあることに注意，エラーの原因が上記の２つのどちらかであるかをよく考えること.

ERR コード：60　Disk　full

意味：指定されたドライブ中のディスケットの記憶領域が一杯になった.

原因：ファイルをたくさんつくって記憶領域が足りなくなった．またはファイルの長さが長くなって入りきらなくなった.

───→ 不要のファイルは kill する．それでもだめならば，新しいディスケットまたは，あまりファイルを作っていないディスケットを使用する.

：PUT#で指定したコード番号が大きすぎる.
BASIC は指定したレコード数だけの領域をとろうとするので，あまり大きな番号を指定すると記憶領域が足らなくなる.

ERR コード：61 Input past end

意味：ファイルの長さを越えて，ファイルを読もうとした.

原因：EOF を越えてINPUT#で入力しようとした.

：GET# で指定したレコード番号がファイルの大きさ（LOFの返す値）を越えている.

ポイント：シーケンシャルファイルでは，INPUT#の前にEOFのチェックを行なうこと．ランダムファイルでは，ファイルをオープンしたならば，LOF でレコード番号の最大値を調べておくこと.

ERR コード：62 Bad file name

意味：指定されたファイル名は，ファイル名として不適切なものである.

原因：ファイル名がヌルストリングである.

：ファイル名中にコントロール文字が含まれている.

ポイント：ファイル名を文字変数や文字式で表わす場合にヌルコードが入ったりすることがあるので注意すること.

ERR コード：63 Direct statement in file

意味：ファイル中にダイレクトステートメントが検出された.

原因：プログラムの LOAD 時に，ダイレクトステートメントが検出された.

———→ プログラムファイルでないファイルをロードしようとしたのではないか.

：プログラムの MERGE中に，ダイレクトステートメントが検出された.

MERGE で指定するファイルは，アスキー型式のプログラムファイルでなければならない.

ERR コード：64 Bad allocation table

意味：ディスケット中の FAT が破壊されている.

原因：DSKO$ でFAT に対して書き込みを行なったり，ディスケットの保存状態が悪く記録が失なわれた等の原因が考えられる．いずれにしてもこのエラーが起きたならば，そのディスケットはフォーマッティングを行なわないと使用不可能である.

ERR コード：65 Bad drive number

意味：ドライブ番号の指定がまちがっている.

原因：ドライブ番号の指定において PC につながっていないドライブの番号を指定している.

———→ ドライブ番号は，ドライブ2台の時は1または2，4台のときは1～4のいずれかでなければならない

ERR コード：66　Bad　track／sector

意味：トラック，セクタ番号の指定がまちがっている．

原因：DSKI$, DSKO$ においてトラック，セクタ番号の指定が存在しないものを指定している．

ポイント：トラック番号は 0 から35まで．

セクタ番号は 1 から16の間でなければならない．

ERR コード：67　Deleted　record

意味：消去済みのレコードを扱おうとした．

原因：通常このエラーは発生しない．

ERR コード：68　Rename　across　disks

意味：NAME のファイル名指定が複数のドライブにまたがっている．

原因：同上．新旧のファイル名の指定は同一のドライブでなければならない．

ERR コード：69　Sequential　after　put

意味：PUT した後でシーケンシャルファイルに対する操作を行なった．

原因：PUTした後，PRINT#，INPUT#を用いた．

PUTの後に一度 GET を行なえばこのエラーはでない．シーケンシャルファイルの操作とランダムファイルの操作を混用しないようにするのが最も好ましい．

ERR コード：70　Sequential　I/O　only

意味：シーケンシャル入出力しか行なってはならない．

原因：通常このエラーは発生しない．

ERR コード：71　File　not　OPEN

意味：指定されたファイルは OPEN されていない．

原因：OPEN のし忘れ．

OPEN するファイル番号の指定まちがい．

ERR コード：72　File　write　protcted

意味：指定されたファイルには書き込み禁止属性（P属性）がつけられている．

原因：P属性のついたドライブ中のファイル，またはP属性のついたファイルを KILL しようとした．

KILL する必要があるときにはまず SET で属性を解除する．

：P属性のついたドライブ，ファイル，ファイル番号に対して，PRINT#または PUT#で書き込みを行なおうとした．

ERR コード：73　Disk　offline

意味：指定されたディスクはつながっていない．

原因：通常このエラーは発生しない．

——APPENDIX　B——

ディスク I／O　ルーチンの呼び方

機械語でディスクに対する入出力を行ない方のために，ディスク入出力ルーチンのエントリポイントとコーリングシーケンスを紹介しておきます．

●INITDS　(00D2H)

コントローラを初期化する．同時にディスクが何台つながっているかを調べてAレジスタに入れて戻る．他のレジスタは破壊されない．

●PSHYDIO　(0099H)

ディスク入出力ルーチン．各レジスタに次の値を設定してからコールする．

B, C：トラック番号をB，セクタ番号をCに入れる．
D, E：データの転送開始アドレスを入れる．
A：入出力を行なうセクタ数を入れる．
PSW：入出力の種類を指定する．

C＝1：Write　data
Z＝0：Read　data
Z＝1：Check　data（ベリファイ）

ドライブの指定は EDC6 番地の内容で行なう．0でドライブ1，3でドライブ4が選択される．また EDC5 番地はエラー回数のカウンタとして使用されているので事前にクリアしておくこと．エラー回数が10回をこえるとエラーメッセージ（Disk I／O error）が出される．リターン時には CY フラグがステータスを示す．

CY＝1ならばエラーが発生したことを示し．

CY＝0ならばなかったことを示す．BC レジスタは破壊されない．

使用例-ドライブ2のトラック0，セクタ1を読み C000 番地以降にロードする．

```
LD      DE, 0C00H
LD      BC, 0001H
LD      A, 1
LD      (0EDC6H), A
OR      A
CALL    0099H
```

——APPENDIX　C——

ディスケットを両面使う方法

普通市販されているディスケットは，裏表ともに，ヘッドアクセスホールがついていますが，PCでは表側，すなわちラベルの貼ってある側しか使用することができません．しかし裏側にも磁性体がコーティングしてありますから，少し加工を加えることにより使用可能となります．

ディスケットの裏側を使用するには次の手続きを施してください．

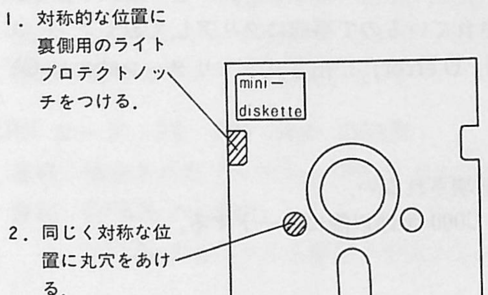
1．裏側にもライト・プロテクト・ノッチをつける．

2．同じくインデックスホール検出用の穴をあける．

3，システムに付属の format プログラムによりフォーマッティングを行なう．

まず1でライトプロテクトノッチをつけないと，ディスケット裏面への書き込みが行なえません．左右対称な位置に正しく穴をあけてください．2のインデックスホール検出内の穴は通常の読み書きの時には不用なのですが，ディスケットのフォーマッティングをする時には必要なので必ずあけてください．穴をあけるのはジャケットだけで，記憶媒体にはあけてはいけません．本体に穴や傷ができると，フォーマッティングが正しく行なえなくなったり，回転がなめらかにならなくなったりしますから注意すること．ジャケットには裏表とも対称な位置に正しく穴をあけること．

穴をあけたならば format プログラムによりフォーマッティングをして，できあがり．表側と同様に使えるはずです．



Disk-BASIC入門
アスキー・システム・バンクPC-8000＃2
©1980 アスキー出版 検印廃止

共　著　飯田陽一、鈴木義明
　　　　大野俊治
監　修　アスキー・ラボラトリーズ
発行者　塚本慶一郎

発行所株式会社　アスキー出版
〒107 東京都港区南青山5－16－1 青山ビル5F
振替 東京7－57496
☎03－407－4910（業務部）03－407－4903（出版部）

1980年5月10日 第1版第2刷発行　Printed in Japan
3B06A0202

本の注文，発送については業務部，書籍の内容については出版部，上記のいずれかに電話して下さい．

アスキー出版